THINK C

The Professional's Choice

Object-Oriented Programming Manual

Credits

User Manual	Philip Borenstein and Jeff Mattson
THINK C Application	Michael Kahl, Greg Howe, Mike Cote, and John McEnerney
THINK Class Library	Dan Podwall and Gregory H. Dow
Quality Assurance	David Allcott, Michael Rockhold, and Paul Vetri
Technical Support	Michael Carland, Mark Geschelin, and Phil Shapiro
Marketing Manager	Susan Smith
Product Manager	Philip Borenstein

Copyright © 1989, 1991 Symantec Corporation.
All Rights Reserved.

Symantec Corporation
10201 Torre Avenue
Cupertino, CA 95014
408/253-9600

THINK C and THINK Pascal are trademarks of Symantec Corporation. Other brands and their products are trademarks of their respective holders.

ResEdit, SARez, and SADeRez are copyrighted programs of Apple Computer, Inc. licensed to Symantec Corp. to distribute for use only in combination with THINK C. Apple software shall not be copied onto another diskette (except for archive purposes) or into memory unless as part of execution of THINK C. When THINK C has completed execution, Apple Software shall not be used by any other program.

The *THINK C Object-Oriented Programming Manual* is copyrighted and all rights reserved. Information in this document is subject to change without notice and does not represent a commitment on the part of Symantec Corporation. The software described in this document is furnished under a license agreement. The document may not, in whole or in part, be copied, photocopied, reproduced, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Symantec Corporation.

SYMANTEC CORPORATION MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, REGARDING THE ENCLOSED COMPUTER SOFTWARE PACKAGE, ITS MERCHANTABILITY, OR ITS FITNESS FOR ANY PARTICULAR PURPOSE. THE EXCLUSION OF IMPLIED WARRANTIES IS NOT PERMITTED BY SOME STATES. THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY PROVIDES YOU WITH SPECIFIC LEGAL RIGHTS. THERE MAY BE OTHER RIGHTS THAT YOU MAY HAVE WHICH VARY FROM STATE TO STATE.

Contents

Getting Started with Objects

1 Welcome . . . 3
- If you're new to object-oriented programming . . . 3
- If you're familiar with object-oriented programming . . . 3
- If you know the THINK Class Library . . . 3

Contents . . . 3
What's in this Manual . . . 5
What's New in the THINK Class Library . . . 6
- Long coordinates . . . 6
- Improved error handling . . . 7
- Dependent objects . . . 7
- Director owners . . . 8
- Abstract text . . . 8
- Compatibility with earlier versions . . . 8

2 Installing theTHINK Class Library . . . 9
Installation . . . 11
Building the Starter Project . . . 12
Installing the TMPL Resources into ResEdit . . . 13

Object-Oriented Programming

3 Object-Oriented Programming . . . 17
Contents . . . 17
Overview . . . 19
Objects and Messages . . . 19
Classes . . . 20
Inheritance and Polymorphism . . . 21
Objects and the Macintosh Interface . . . 22
Working with Objects . . . 23
- What classes should I define? . . . 23
- When should I create a subclass? . . . 23
- What should be a method? . . . 24
- When should I use procedural programming? . . . 24

Where to Go Next . . . 24

4 Using Objects in THINK C . . . 25
How to read this chapter . . . 25
Contents . . . 25
Introduction . . . 27
What you should know . . . 27
A note on terminology . . . 27
Using the Object Extensions . . . 28
Overview . . . 28
Declaring a Class . . . 30
Declaring root classes . . . 30
Declaring subclasses . . . 31
Declaring and Using Objects . . . 32
Creating and deleting objects . . . 32
Assignment and objects . . . 33
Objects and memory . . . 34
Accessing Members of a Class . . . 35
Declaring Members of a Class . . . 36
Declaring instance variables . . . 37
Declaring class variables . . . 38
Declaring methods . . . 39
Virtual and non-virtual methods . . . 39
Class methods . . . 40
Examples of methods . . . 40
Declaring circular and forward references . . . 41
Defining and Using Methods . . . 42
Access Control . . . 43
Using access specifiers . . . 44
Access specifiers in subclass declarations . . . 45
Protected access . . . 46
Constructors and Destructors . . . 48
Declaring and defining constructors . . . 48
How constructors are invoked . . . 49
Using access specifiers with constructors . . . 50
Destructors . . . 50
Allocators and Deallocators . . . 51
Allocators . . . 52
Deallocators . . . 53
Default allocator and deallocator . . . 54
Tips and Techniques . . . 54
Organizing your classes . . . 55
Objects and the Macintosh . . . 55
Direct and indirect classes . . . 58
Object Extension Options . . . 58
Turning on the object extensions . . . 58
Classes are indirect by default . . . 59
Methods are virtual by default . . . 59
Optimize monomorphic methods . . . 59
Special Functions . . . 59

Differences between THINK C and C++ . . . 60
General differences . . . 60
Keywords . . . 61
Value objects and reference objects . . . 61
THINK C extensions . . . 62
Access declarations, friends, and multiple inheritance . . . 62
Constructors and destructors . . . 62
Scope resolution operator . . . 62
Sizeof operator not allowed . . . 62

5 Tutorial: LearnOOP . . . 63
Before you begin . . . 63
What you should know . . . 63
Contents . . . 64
What Does LearnOOP Do? . . . 65
How Does LearnOOP Work? . . . 67
How Do Objects Know How to Draw Themselves? . . . 68
The LearnOOP Class Declarations . . . 71
How Do You Create an Object? . . . 74
How Do You Define a New Class? . . . 75
But How Does It Know? . . . 77
Where to Go Next . . . 78

6 The Class Browser . . . 79
Contents . . . 79
Using the Class Browser . . . 81
Finding the Declaration of a Class . . . 81
Finding a class declaration with the Class Browser . . . 81
Finding a class declaration with the editor . . . 82
Finding the Definition of a Method . . . 83
Finding a method definition with the Class Browser . . . 83
Finding a method definition with the editor . . . 84
Keyboard Shortcuts in the Class Browser . . . 85
Class Browser Summary . . . 86

The THINK Class Library

7 The THINK Class Library . . . 89
Contents . . . 89
Introduction . . . 91
What you should know . . . 91
Conventions . . . 91
Types . . . 91
Naming conventions . . . 92
Object-oriented terminology . . . 92

Variables . . . 165
Global variable . . . 166
Event related instance variables . . . 166
Phase related instance variables . . . 166
Memory related instance variables . . . 167
Standard file instance variables . . . 168
Methods . . . 168
Initialization methods . . . 168
Advanced initialization methods . . . 173
Accessing methods . . . 174
Command methods . . . 174
Memory management methods . . . 177
Execution methods . . . 180
Document methods . . . 183
Periodic task methods . . . 184
Class resources . . . 184

12 CArray . . . 185
Introduction . . . 185
Heritage . . . 185
Using CArray . . . 185
Variables . . . 187
Methods . . . 187
Creation and destruction methods . . . 187
Accessing methods . . . 187
Insertion and deletion methods . . . 188
Membership method . . . 188
Moving methods . . . 189
Resizing methods . . . 190
Temporary storage methods . . . 190
Offset method . . . 190

13 CBartender . . . 191
Introduction . . . 191
Heritage . . . 191
Using CBartender . . . 191
Creating standard menus . . . 191
Resource based menus . . . 194
Creating hierarchical menus . . . 196
Dimming and checking menu items . . . 197
Variables . . . 198
Global variable . . . 198
Instance variables . . . 198
Methods . . . 198
Construction methods . . . 199
Insertion and deletion methods . . . 199
Item manipulation methods . . . 200
Item insertion and deletion . . . 201
Look-up methods . . . 201
Appearance methods . . . 202
Command Extraction methods . . . 204

14 CBitMap . . . 205
Introduction . . . 205
Heritage . . . 205
Using CBitMap . . . 205
Methods . . . 207
Construction and destruction methods . . . 207
Accessing methods . . . 207
Image copying methods . . . 208
Drawing preparation methods . . . 208

15 CBitMapPane . . . 209
Introduction . . . 209
Heritage . . . 209
Using CBitMapPane . . . 209
Variables . . . 210
Methods . . . 210
Construction and destruction methods . . . 210
Accessing methods . . . 211
Drawing method . . . 211

16 CBureaucrat . . . 213
Introduction . . . 213
Heritage . . . 213
Using CBureaucrat . . . 213
Variables . . . 214
Global variable . . . 214
Instance variable . . . 214
Methods . . . 214
Construction and destruction methods . . . 214
Accessing method . . . 214
Command methods . . . 214
Change notification methods . . . 217

17 CButton . . . 219
Introduction . . . 219
Heritage . . . 219
Using CButton . . . 219
Variables . . . 220
Methods . . . 220

18 CCharGrid . . . 223
Introduction . . . 223
Heritage . . . 223
Using CCharGrid . . . 223
Variables . . . 225
Methods . . . 225
Construction methods . . . 225
Drawing methods . . . 226

19 CCheckBox . . . 227
Introduction . . . 227
Heritage . . . 227
Using CCheckBox . . . 227
Variables . . . 228
Methods . . . 228

20 CChore . . . 231
Introduction . . . 231
Heritage . . . 231
Using CChore . . . 231
Idle chores . . . 231
Urgent chores . . . 232
Using chores . . . 232
Variables . . . 232
Methods . . . 232

21 CClipboard . . . 235
Introduction . . . 235
Heritage . . . 235
Using CClipboard . . . 235
Implementing a CClipboard subclass . . . 236
Variables . . . 236
Global variable . . . 236
Instance variables . . . 237
Methods . . . 237
Construction and destruction methods . . . 237
Suspend and resume methods . . . 238
Appearance methods . . . 238
Accessing methods . . . 239
Scrap conversion methods . . . 240
Class resources . . . 242

22 CCluster . . . 243
Introduction . . . 243
Heritage . . . 243
Using CCluster . . . 243
Variables . . . 244
Methods . . . 244
Construction and destruction . . . 244
Insertion and deletion . . . 244
Membership . . . 245
Iteration . . . 246

23 CCollaborator . . . 249
Introduction . . . 249
Heritage . . . 249
Using CCollaborator . . . 249
Variables . . . 252

Methods . . . 252
Creation and destruction . . . 252
Creating dependency method . . . 253
Change notification methods . . . 253
List update methods . . . 253

24 CCollection . . . 255
Introduction . . . 255
Heritage . . . 255
Using CCollection . . . 255
Variables . . . 255
Methods . . . 255

25 CControl . . . 257
Introduction . . . 257
Heritage . . . 257
Using CControl . . . 257
Variables . . . 257
Methods . . . 258
Construction and destruction methods . . . 258
Accessing methods . . . 259
Appearance methods . . . 261
Click response methods . . . 263

26 CDataFile . . . 263
Introduction . . . 263
Heritage . . . 263
Using CDataFile . . . 263
Variables . . . 264
Methods . . . 264
Construction and destruction methods . . . 264
Accessing methods . . . 265
Open and close methods . . . 265
Read and write methods . . . 267

27 CDecorator . . . 267
Introduction . . . 267
Heritage . . . 267
Using CDecorator . . . 267
Variables . . . 267
Global variable . . . 268
Instance variables . . . 268
Methods . . . 269

28 CDesktop . . . 269
Introduction . . . 269
Heritage . . . 269
Using CDesktop . . . 269

Variables . . . 269
Global variable . . . 269
Instance variables . . . 270
Methods . . . 270
Construction and destruction methods . . . 270
Appearance methods . . . 270
Mouse methods . . . 271
Window methods . . . 273
Accessing methods . . . 274
Calibration methods . . . 274
Cleanup method . . . 274

29 CDirector . . . 275
Introduction . . . 275
Heritage . . . 275
Using CDirector . . . 275
Variables . . . 276
Global variable . . . 276
Instance variables . . . 276
Methods . . . 276
Creation and destruction . . . 276
Accessing method . . . 277
Command methods . . . 277
Appearance methods . . . 278
Window methods . . . 279
Change notification method . . . 280

30 CDirectorOwner . . . 281
Introduction . . . 281
Heritage . . . 281
Using CDirectorOwner . . . 281
Variables . . . 282
Methods . . . 282
Creation and destruction . . . 282
Insertion and deletion methods . . . 282
Appearance methods . . . 282

31 CDocument . . . 285
Introduction . . . 285
Heritage . . . 285
Using CDocument . . . 285
Variables . . . 287
Global variables . . . 287
Instance variables . . . 287

Methods . . . 288
Construction and destruction methods . . . 288
Command methods . . . 288
Appearance Methods . . . 290
File Creation . . . 290
Printing Methods . . . 291
Filing Methods . . . 292
Undo methods . . . 293
Class Resources . . . 293

32 CEditText . . . 295
Introduction . . . 295
Heritage . . . 295
Using CEditText . . . 295
Variables . . . 296
Methods . . . 296
Construction and destruction methods . . . 296
Mouse and Keystrokes methods . . . 297
Command methods . . . 297
Display methods . . . 297
Text specification methods . . . 298
Text characteristics methods . . . 299
Calibrating methods . . . 301
Printing methods . . . 302
Cursor methods . . . 302

33 CEnvironment . . . 305
Introduction . . . 305
Heritage . . . 305
Using CEnvironment . . . 305
Variables . . . 305
Methods . . . 305

34 CError . . . 307
Introduction . . . 307
Heritage . . . 307
Using CError . . . 307
Variables . . . 307
Global variable . . . 307
Instance variables . . . 308
Methods . . . 308
Error reporting methods . . . 308
Functions . . . 308
Class resources . . . 309

35 CFile . . . 311
Introduction . . . 311
Heritage . . . 311
Using CFile . . . 311

Variables . . . 312
Methods . . . 312
Construction and destruction methods . . . 312
Specifying methods . . . 312
Open and close methods . . . 313
Accessing methods . . . 313
Filing methods . . . 313

36 CFWDesktop . . . 315
Introduction . . . 315
Heritage . . . 315
Using CFWDesktop . . . 315
Variables . . . 316
Methods . . . 316
Construction and destruction methods . . . 316
Appearance methods . . . 316
Mouse methods . . . 317
Window methods . . . 317
Cleanup method . . . 318

37 CGridSelector . . . 319
Introduction . . . 319
Heritage . . . 319
Using CGridSelector . . . 319
Variables . . . 319
Methods . . . 320
Construction methods . . . 320
Drawing methods . . . 320
Accessing methods . . . 321

38 CList . . . 323
Introduction . . . 323
Heritage . . . 323
Using CList . . . 323
Variables . . . 323
Methods . . . 323
Construction methods . . . 323
Insertion and deletion methods . . . 324
Ordering methods . . . 324
Membership methods . . . 325

39 CMBarChore . . . 327
Introduction . . . 327
Heritage . . . 327
Using CMBarChore . . . 327
Variables . . . 327
Methods . . . 327

40 CMenuDefProc . . . 329
Introduction . . . 329
Heritage . . . 329
Using CMenuDefProc . . . 329
Creating the stub MDEF . . . 329
Writing the CMenuDefProc subclass . . . 330
Using your CMenuDefProc subclass . . . 330
Examples of MDEF objects . . . 331
Variables . . . 332
Methods . . . 332
Functions . . . 334

41 CMouseTask . . . 335
Introduction . . . 335
Heritage . . . 335
Using CMouseTask . . . 335
Defining a mouse task . . . 335
Using the mouse task . . . 336
Variables . . . 337
Methods . . . 337
Construction and destruction methods . . . 337
Mouse tracking methods . . . 337
Class resources . . . 338

42 CObject . . . 339
Introduction . . . 339
Heritage . . . 339
Using CObject . . . 339
Variables . . . 339
Methods . . . 339
Creation and destruction . . . 340

43 CPane . . . 341
Introduction . . . 341
Heritage . . . 341
Using CPane . . . 341
Coordinate Systems in Panes . . . 342
Drawing in Panes . . . 344
Variables . . . 346
Methods . . . 347
Construction/Destruction methods . . . 347
Accessing methods . . . 349
Appearance methods . . . 351
Size and location methods . . . 352
Adapting methods . . . 353
Drawing methods . . . 354
Printing methods . . . 355
Calibration methods . . . 356
Cursor method . . . 357
Coordinate transformation methods . . . 357

44 CPaneBorder . . . 361
Introduction . . . 361
Heritage . . . 361
Using CPaneBorder . . . 361
Variables . . . 363
Methods . . . 364
Creation and destruction . . . 364
Accessing methods . . . 364
Drawing method . . . 365
Calibration method . . . 366

45 CPaneMDEF . . . 367
Introduction . . . 367
Heritage . . . 367
Using CPaneMDEF . . . 367
Variables . . . 368
Methods . . . 368
Construction and destruction methods . . . 368

46 CPanorama . . . 371
Introduction . . . 371
Heritage . . . 371
Using CPanorama . . . 371
Variables . . . 372
Methods . . . 372
Construction and destruction methods . . . 372
Accessing methods . . . 373
Calibrating methods . . . 375
Scrolling methods . . . 375
Printing methods . . . 376

47 CPatternGrid . . . 377
Introduction . . . 377
Heritage . . . 377
Using CPatternGrid . . . 377
Variables . . . 379
Methods . . . 379
Construction methods . . . 379
Drawing methods . . . 380
Accessing methods . . . 380

48 CPictFile . . . 381
Introduction . . . 381
Heritage . . . 381
Using CPictFile . . . 381
Variables . . . 381
Methods . . . 381

49 CPicture . . . 383
Introduction . . . 383
Heritage . . . 383
Using CPicture . . . 383
Variables . . . 383
Methods . . . 384
Construction/Destruction . . . 384
Appearance methods . . . 385
Accessing methods . . . 385
Calibration methods . . . 385

50 CPNTGFile . . . 387
Introduction . . . 387
Heritage . . . 387
Using CPNTGFile . . . 387
Variables . . . 387
Methods . . . 387

51 CPrinter . . . 389
Introduction . . . 389
Heritage . . . 389
Using CPrinter . . . 389
Variables . . . 391
Methods . . . 391
Construction and destruction methods . . . 391
Accessing methods . . . 392
Printing methods . . . 394

52 CRadioControl . . . 397
Introduction . . . 397
Heritage . . . 397
Using CRadioControl . . . 397
Variables . . . 397
Methods . . . 398

53 CRadioGroupPane . . . 399
Introduction . . . 399
Heritage . . . 399
Using CRadioGroupPane . . . 399
Variables . . . 400
Methods . . . 400
Construction and destruction methods . . . 400
Accessing methods . . . 401
Change notification method . . . 401

54 CResFile . . . 403
Introduction . . . 403
Heritage . . . 403

Using CResFile . . . 403
Variables . . . 403
Methods . . . 403

55 CRunArray . . . 405
Introduction . . . 405
Heritage . . . 405
Using CRunArray . . . 405
Variables . . . 406
Methods . . . 406
Creation method . . . 406
Accessing method . . . 406
Insertion and deletion methods . . . 406
Membership method . . . 407
Summing methods . . . 407
Run-handling methods . . . 407

56 CScrollBar . . . 409
Heritage . . . 409
Using CScrollBar . . . 409
Variables . . . 410
Methods . . . 410
Construction and destruction methods . . . 410
Accessing methods . . . 410
Drawing methods . . . 411
Click response methods . . . 411

57 CScrollPane . . . 413
Introduction . . . 413
Heritage . . . 413
Using CScrollPane . . . 413
Variables . . . 414
Methods . . . 414
Construction and destruction methods . . . 414
Accessing methods . . . 415
Scroll bar maintenance methods . . . 416
Scroll performance methods . . . 416
Functions . . . 417

58 CSelector . . . 419
Introduction . . . 419
Heritage . . . 419
Using CSelector . . . 419
Variables . . . 420
Methods . . . 420
Construction and destruction methods . . . 420
Mouse methods . . . 421
Accessing methods . . . 421

59 CSelectorMDEF . . . 425
Introduction . . . 425
Heritage . . . 425
Using CSelectorMDEF . . . 425
Variables . . . 425
Methods . . . 425
Construction and destruction methods . . . 425

60 CSizeBox . . . 427
Introduction . . . 427
Heritage . . . 427
Using CSizeBox . . . 427
Variables . . . 427
Methods . . . 427
Construction and destruction methods . . . 427
Appearance methods . . . 428
Class resources . . . 428

61 CStack . . . 429
Introduction . . . 429
Heritage . . . 429
Using CList . . . 429
Variables . . . 429
Methods . . . 429

62 CSwitchboard . . . 431
Introduction . . . 431
Heritage . . . 431
Using CSwitchboard . . . 431
Variables . . . 431
Methods . . . 432
Initialization methods . . . 432
Mouse methods . . . 432
Key methods . . . 432
Disk methods . . . 432
Window event methods . . . 433
Suspend/Resume methods . . . 433
Event processing methods . . . 433

63 CTask . . . 437
Introduction . . . 437
Heritage . . . 437
Using CTask . . . 437
Variables . . . 438
Methods . . . 438
Initialization methods . . . 438
Accessing methods . . . 438
Action methods . . . 439
Class resources . . . 439

64 CTearChore . . . 441
Introduction . . . 441
Heritage . . . 441
Using CTearChore . . . 441
Variables . . . 441
Methods . . . 441

65 CTearOffMenu . . . 443
Introduction . . . 443
Heritage . . . 443
Using CTearOffMenu . . . 443
Variables . . . 444
Methods . . . 444

66 CTextEditTask . . . 447
Introduction . . . 447
Heritage . . . 447
Using CTextEditTask . . . 447
Variables . . . 448
Methods . . . 448
Creation and destruction methods . . . 448
Accessing method . . . 449
Action methods . . . 449
Internal methods . . . 450

67 CTextEnvirons . . . 453
Introduction . . . 453
Heritage . . . 453
Using CTextEnvirons . . . 453
Variables . . . 454
Methods . . . 455

68 CTextStyleTask . . . 457
Introduction . . . 457
Heritage . . . 457
Using CTextStyleTask . . . 457
Variables . . . 458
Methods . . . 458
Construction method . . . 458
Action methods . . . 458
Internal methods . . . 459

69 CView . . . 461
Introduction . . . 461
Heritage . . . 461
Using CView . . . 461
Views and the visual hierarchy . . . 461
Views and the chain of command . . . 462
Using Balloon Help with views . . . 462
Variables . . . 463
Methods . . . 463
Construction and destruction methods . . . 463
Accessing methods . . . 464
Appearance methods . . . 466
Mouse methods . . . 467
Cursor methods . . . 468
Subview Management methods . . . 471
Class resources . . . 473

70 CWindow . . . 475
Introduction . . . 475
Heritage . . . 475
Using CWindow . . . 475
Variables . . . 476
Global variable . . . 476
Instance variables . . . 476
Methods . . . 476
Construction and destruction methods . . . 476
Accessing methods . . . 478
Appearance methods . . . 480
Size and location methods . . . 481
Drawing methods . . . 482
Mouse methods . . . 483
Conversion methods . . . 483

71 TCL Library Routines . . . 485
Introduction . . . 485
Toolbox Utilities . . . 485
QuickDraw Utilities . . . 485
Window Manager Utilities . . . 486
Dialog Manager Utilities . . . 486
Font Manager Utility . . . 487
Keyboard Utilities . . . 487
String Utilities . . . 487
Operating System Utilities . . . 488
Long Coordinate Utilities . . . 488
Long point utilities . . . 489
Long rectangle utilities . . . 490
THINK Class Library Utilities . . . 491
Error reporting utility . . . 492
Memory allocation utilities . . . 492
Memory disposal utilities . . . 494

Long Coordinate QuickDraw Utilities 494
Long rectangle drawing routines 495
Long oval drawing routines 495
Long rounded rectangle drawing routines 496
Long bit transfer routine 497
Long point pen routines 497
Long region utility routines 497

72 Global Variables 499
Introduction 499
Global Objects 499
Mouse Click Globals 500
Cursors 501
System Globals 501
Error Globals 503
Utility Globals 503

Index 505

THINK C

Getting Started with Objects

Part One

1 Welcome
2 Installing the THINK Class Library

Welcome 1

This manual is about object-oriented programming in THINK C. It describes the object extensions to C and the THINK Class Library. The THINK Class Library is a collection of classes that define a generic Macintosh application.

If you're new to object-oriented programming
You should read this chapter and Chapter 3, "Object-Oriented Programming." Then read the first part of Chapter 4, "Using Objects in THINK C," and follow the tutorial in Chapter 5, "Tutorial: LearnOOP."

If you're familiar with object-oriented programming
Read this chapter, Chapter 4, "Using Objects in THINK C," and Chapter 7, "The THINK Class Library."

If you know the THINK Class Library
Be sure to read "What's New in the THINK Class Library" on page 6 in this chapter. Then read Chapter 4, "Using Objects in THINK C to learn about the enhanced object extensions to THINK C. You should scan though Chapter 7, "The THINK Class Library," and read Chapter 8, "Exception Handling."

Contents

What's in this Manual 5
What's New in the THINK Class Library 6
Long coordinates 6
Improved error handling 7
Dependent objects 7
Director owners 8
Abstract text 8
Compatibility with earlier versions 8

What's in this Manual

This manual is organized in three sections: Getting Started, Object-Oriented Programming in THINK C, The THINK Class Library.

Getting Started

This is the section you're reading. It contains this chapter and instruction for installing the THINK Class Library.

Object-Oriented Programming

This section describes object-oriented programming with THINK C.

Object-Oriented Programming	This chapter is a general introduction to object-oriented programming. You'll learn the terminology and basic concepts of object-oriented programming.
Using Objects in THINK C	This chapter describes how THINK C implements objects. You should read this chapter even if you've used objects in THINK C before.
Tutorial: LearnOOP	LearnOOP is a hands-on demonstration of the basic concepts of object-oriented programming.
The Class Browser	This chapter shows you how to use the THINK C Class Browser. The browser is a tool that lets you examine your classes and look up definitions of instance variables and methods.

The THINK Class Library

This section contains 66 chapters that describe the THINK Class Library and each class in the library. This section is designed for both THINK C and THINK Pascal programmers.

The THINK Class Library	This chapter introduces you to the class library. It gives you an overview, and tells you how the different parts of the THINK Class

	Library fit together. Be sure to read this chapter even if you've used the THINK Class Library before.
Exception Handling	This chapter describes the exception handling mechanism used in the THINK Class Library. The exception handling mechanism lets you handle errors in a consistent and graceful way.
Chapter 9–Chapter 70	These chapters describe each class in the THINK Class Library.
TCL Library Routines	This chapter describes library routines used throughout the THINK Class Library.
Global Variables	This chapter lists all of the global variables that the THINK Class Library uses.

What's New in the THINK Class Library

This version of the THINK Class Library is version 1.1. If you've used earlier versions of the THINK Class Library, you'll find that the fundamental model behind the TCL has not changed. The visual hierarchy and the chain of command work just as they did before. This section describes the most important new features of this version of the THINK Class Library and how they affect your existing programs.

Long coordinates

The visual hierarchy now uses 32-bit (long) coordinates. This change means that your panes and panoramas are no longer limited to the QuickDraw coordinate space. You don't have to use 32-bit coordinates. In fact, the default is to make turn long coordinates off. However, this change will require you to modify your programs.

The THINK Class Library uses the types LongRect and LongPt throughout the CView family. These types are defined like this in C:

```
typedef struct LongPt
{
   long v, h;
} LongPt;

typedef struct LongRect
{
   long top, left, bottom, right;

} LongRect;
```

And like this in Pascal:

```
type
   LongPt = record
               case integer of
            1: (  v, h: longint
               );
            2: (  vh: array[VHSelect] of longint
               );
            end;

   LongRect = record
               case integer of
            1: (  top, left,
                  bottom, right: longint
               );
            2: (  topLeft: LongPt;
                  botRight: LongPt
               );
            end;
```

Please be sure to read "Working with Panes" on page 103 and the description of CPane. To help make the transition easier, you should also read "Long Coordinate Utilities" on page 488.

Improved error handling

The TCL now uses an exception handling mechanism to catch errors. This mechanism lets you separate the normal flow of your code from the error handling code. Most of the classes in the THINK Class Library use the exception handling mechanism. Be sure to read Chapter 8, "Exception Handling." For a good example of exception handling, look at the CDataFile class.

Dependent objects

All of the chain of command classes are now descendants of a new class called CCollaborator. This class lets you set up dependencies between ob-

jects, so one object can announce a change to its dependents. Read the chapter on CCollaborator.

Director owners

In earlier versions of the THINK Class Library, only an application could a director's supervisor. This version introduces a new class, CDirectorOwner, for objects that own directors. CApplication and CDirector are director owners. This class lets you create multi-window documents by creating directors that are supervised by other directors or documents.

Abstract text

The class CAbstractText defines characteristics of text classes. The class CStaticText has been removed. Instead, you specify whether text is editable, styleable, or selectable. See the chapters on CEditText and CAbstractText.

Compatibility with earlier versions

Older versions of the classes that have changed substantially in this version of the THINK Class Library are in the `Compatibility classes` folder.

Old Class	Replaced because...
CBarOwner	The functionality that CBarOwner provided is now part of CBartender.
CBorder	Borders are now handled by CPaneBorder which is not a subclass of CPane.
CColorWindow	This functionality is now in CWindow.
CRadioButton CRadioGroup	The new class CRadioGroupPane is a pane that groups radio controls. CRadioControl replaces CRadioButton.
CStaticText	All text classes are now descendants of CAbstractText. For static text, use CEditText, but specify that it is not editable.
CCluster CList	These classes are now descendants of CArray. The old classes are named OCluster and OList.
CFile CDataFile	These classes now use the exception handling mechanism instead of returning error codes. The old classes are named OFile and ODataFile.

Installing the THINK Class Library 2

This chapter tells you how to install the THINK Class Library on your disk. The THINK Class Library is a set of classes that implement a generic Macintosh application. Chapter 7, "The THINK Class Library," talks about the THINK Class Library in detail.

You don't need to use the THINK Class Library to use THINK C's object extensions. If you just want to learn how to useobject-oriented programming with THINK C, read Chapter 3, "Object-Oriented Programming," and Chapter 4, "Using Objects in THINK C."

Contents

Installation 11
Building the Starter Project 12
Installing the TMPL Resources into ResEdit 13

Installation

Before you install the THINK Class Library on your disk, be sure that you followed the instructions in *THINK C User Manual*, Chapter 2, "Installing THINK C." Also, make sure you have enough space on your hard disk. The files that make up the core of the THINK Class Library take up nearly 1300K of disk space.

Note

You'll want to install the THINK Class Library demos even if you're familiar with the previous version of the THINK Class Library. The demos show you how to use new classes and take advantage of changes in old classes.

To install the THINK Class Library, run the self-extracting archive `THINK Class Library 1.1.sea`, and choose your `Development` folder as the destination folder. This archive creates a folder called `THINK Class Library 1.1` and puts these files and folders in it:

This file or folder...	Contains...
`Core classes`	The basic classes all applications need.
`TCL Libraries`	Utility routines.
`Control classes`	Classes for buttons and pop-up menus.
`Dialog classes`	Classes for dialogs.
`File classes`	Classes for specific kinds of files. CFile is in `Core classes`.
`FW/Tearoffs`	Classes for floating windows and tear-off menus.
`Table classes`	Classes for tables, like spreadsheets and lists.
`Text classes`	Classes for editing and displaying text.
`More classes`	Classes that don't fall into any of the above categories, including CBitMap, CStack and CTextEnvirons.
`Compatibility classes`	Classes from THINK Class Library 1.0 that have been replaced with new classes.

This file or folder...	Contains...
`TCL 1.1 doc`	Documentation on some parts of the THINK Class Library, including classes for dialogs and tables.
`TCL Resources`	All the standard resources you need to use the THINK Class Library.
`TCL TMPLs`	A set of `TMPL` resources that make it easy to create resources for pane initialization. The next set of instructions tell you how to install these `TMPL`s into ResEdit.

To install the demos for the THINK Class Library, run the self-extracting archive `TCL 1.1 Demos.sea`, and choose your `Development` folder as the destination folder. This archive creates a folder called `TCL 1.1 Demos` and puts folders in it:

This demo in this folder...	Is...
`Art Class Demo`	A drawing program that uses tear-off menus and floating windows.
`NewClassDemo Folder`	A demonstration of some of the THINK Class Library's new features, including classes for dialogs, tables, pop-up menus, and styled text editing
`Starter Folder`	A minimal project that you can use as a starting point for your own project. It just open and closes empty windows.
`TinyEdit Folder`	A small text editor based on the Starter project.

Building the Starter Project

You'll use the `Starter.π` project as the basis for all your THINK Class Library-based applications. If you build this project now you'll save a lot of time later since THINK C will only need to recompile the new files that you add to the project.

THINK C stores the compiled code for all your source files in the project document, so when you duplicate the `Starter Folder` to create your own projects, the THINK Class Library files will be already compiled. Since building can take anywhere from five minutes on a Macintosh IIci to over half an hour on a Macintosh Plus, you can save yourself a lot of time later by spending a few minutes building the projects now.

The built project requires about 750K of disk space.

To build the `Starter.π` project, follow these steps:

1. Open the project `Starter.π`, in the `Starter Folder`.
2. Choose the **Bring Up to Date** command from the **Project** menu. THINK C loads all the libraries and compiles all the source files.
3. Choose **Quit** from the **File** menu to return to the Finder.

Installing the TMPL Resources into ResEdit

If you use ResEdit to create and edit your resource files, you should install the TMPL resources from `TCL TMPLs` into your copy of ResEdit.

The TMPL resources are resource editor templates for ResEdit. After you install them, it will be easy for you to create certain THINK Class Library resources with ResEdit. Follow these steps to install the `TMPL` resources into ResEdit.

1. Make a duplicate copy of ResEdit.
2. Double-click on the duplicate copy of ResEdit you just made.
3. Open the file `TCL TMPLs`.
4. Select the item `TMPL`.
5. Choose **Copy** from the **Edit** menu.
6. Open the original copy of ResEdit.
7. Choose **Paste** from the **Edit** menu.
8. Choose **Quit** from the **File** menu. When ResEdit asks you to confirm the changes, click on the Yes button.
9. Delete your copied version of ResEdit.

ResEdit is included in your THINK C package in the `THINK C 5.0 Utilitiles.sea` archive. To install it, follow the instructions in the *THINK C User Manual*, Chapter 2, "Installing THINK C."

THINK C

Object-Oriented Programming

Part Two

3 Object-Oriented Programming
4 Using Objects in THINK C
5 Tutorial: LearnOOP
6 The Class Browser

Object-Oriented Programming 3

Object-oriented programming is not hard to learn. In fact, the hardest thing about object-oriented programming is unlearning what you already know about procedural (traditional) programming. To make object-oriented programming easier to learn, this chapter uses examples and comparisons to procedural programming. The examples should make some concepts more concrete, and the comparisons to procedural programming will help you relate what you're learning to something you already know.

Contents

Overview 19
Objects and Messages 19
Classes 20
Inheritance and Polymorphism 21
Objects and the Macintosh Interface 22
Working with Objects 23
What classes should I define? 23
When should I create a subclass? 23
What should be a method? 24
When should I use procedural programming? 24
Where to Go Next 24

Overview

The basic difference between procedural and object-oriented programming is in the way the two disciplines treat data and action. In procedural programming, data and action are two separate things. You define your data structures, and then you define some routines to operate on them. For each data structure you define, you need a new set of routines.

In object-oriented programming, action and data are closely coupled. When you define your data— your objects— you also define their actions. Instead of a set of routines that do something to data, you have a set of objects interacting with each other.

Objects and Messages

An **object** is an entity that contains some data and an associated set of actions that operate on the data. To make an object perform one of its actions, you send it a **message**.

For example, you might create an object that represents a rectangle. Its data contains the locations of the rectangle's four corner points, and its actions might include drawing, erasing, and moving. To draw a rectangle, you send the rectangle a draw message.

Note

For the time being, don't worry about *how* to send a message to an object. The important thing is to start thinking of sending messages as requesting an object to do something.

Contrast this with the way you'd do the same thing in procedural programming. First you define a record that represents the four corners of the data structures, and then you define three routines to draw, to erase, and to move a rectangle. Each of the routines would take a rectangle data structure as an argument.

So the first big advantage of object-oriented programming over procedural programming is that you can keep the routines that operate on a data structure together with the data structure they're operating on. This "keeping together" is called **encapsulation**.

Classes

Encapsulation is a minor advantage of object-oriented programming. A more significant advantage is that objects can inherit data and behavior from other objects.

Every object belongs to a **class**, which defines the implementation of a particular kind of object. A class describes the object's data and the messages it responds to.

Classes are very much like record declarations. You define the private data for the class the same way as you would the fields of a record. In classes, though, the fields are called **instance variables**. Each instance of a class, each object, has its own instance variables just as each variable of a record type has the same fields.

When you send an object a message, it invokes a routine that implements that message. These routines are called **methods**. The class definition includes the method implementations.

One important thing to keep in mind is that message and method are not the same thing. A message is what you send to an object. How an object responds to a message is the method.

You can think of a class as a template for creating objects. It describes an object's data and the messages it responds to. An object is called an **instance** of a class. You can also say that an object is a **member** of a class.

The rest of this chapter uses pictures like this to represent classes. This is what the rectangle class in the example above looks like:

Class
Rectangle

Instance variables
top
left
bottom
right

Messages	**Methods**
Draw	draw line from point to point
Erase	erase lines
Move	offset points

Inheritance and Polymorphism

You can define a class in terms of an existing class. The new class is called the **subclass**, and the existing class is called the **superclass**. A class without a superclass is said to be a **root class**. A subclass inherits all the instance variables and methods of its superclass. Subclasses can define additional instance variables and methods. Subclasses can also **override** methods defined by the superclass.

Overriding a method means that the subclass responds to the same message as its superclass, but it uses its own method to respond to the message.

For example, suppose you want to create a class to represent employees. Two of the instance variables might be the person's name and birth date. You might define three methods, GetName, GetAge and GetWkPay, to return the person's name, to calculate the person's age from the birth date, and to return the person's weekly salary.

Class
Employee

Superclass
none

Instance variables
Name
Birthdate

Messages	**Methods**
GetName	return Name
GetAge	return (Today - Birthdate)
GetWkPay	return 0

Now you can use the Employee class to create two new subclasses: HourlyEmployee and ExemptEmployee. Both of these classes inherit all the instance variables from the Employee class. Each class, however, defines a new instance variable to store the salary, and each class overrides the GetWkPay method to return the appropriate weekly salary.

This is what the HourlyEmployee class looks like.

Class
HourlyEmployee

Superclass
Employee

Instance variables
HourlySalary

Messages	**Methods**
GetWkPay	return (HourlySalary * 40)

And this is what the ExemptEmployee class looks like.

Class
ExemptEmployee

Superclass
Employee

Instance variables
YearlySalary

Messages	**Methods**
GetWkPay	return (YearlySalary / 52)

As long as your object is a member of any of the Employee classes, you can send it GetName, GetAge, and GetWkPay messages, and the object will respond properly.

This ability to send the same message to objects of different classes is called **polymorphism**. If you need to create a new kind of employee (a salesperson on commission), all you need to do is define a subclass of Employee and override the GetWkPay method. Any routine that sends GetWkPay messages to employee objects would still work.

Consider for a moment how you would do the same thing in procedural programming. Your employee record would need a flag field to specify whether it was an HourlyEmployee or an exempt employee. Then, your GetWkPay function would check this field and calculate the weekly salary appropriately. If you added a salesperson, you would have to change the GetWkPay function to handle the salary calculation for the new kind of employee.

Objects and the Macintosh Interface

The Macintosh interface lends itself to object-oriented programming. For instance, in the Finder, the effect of the **Open** command depends on the icons you've selected. If you select a folder, it opens the folder. If you select an application it launches the application. If you select a document, it launches the application and tells it to open the selected document. What you're doing is sending an Open message to different Finder objects. Each object uses its own Open method to carry out the request.

Think of the visual entities on the Macintosh screen—windows, controls, the menu bar, icons—as objects. They're things that are waiting to do something. Think of your actions—clicking, dragging, typing—as messages. You're sending messages to the Macintosh objects.

The THINK Class Library, included with your THINK C package, provides a set of classes that let you work with these Macintosh elements from an object-oriented point of view. You can read more about the THINK Class Library in Chapter 7, "The THINK Class Library."

Working with Objects

If you've never worked with object-oriented programming, this section will give you some general guidelines for designing classes. As you get more comfortable with object-oriented programming, you'll build up a collection of classes you can use in any of your applications. If you want to see how extensive a library of classes can be, look at the THINK Class Library.

What classes should I define?

Usually, you define only one root-level class and then define all the other classes as descendants of that class. The advantage of this approach is that you can define behavior that applies to all the objects in your application.

This root-level class is an example of an abstract class. You don't create instances of abstract classes. Instead, you use them to give a family of objects common behavior. The Employee class in the example above is an abstract class. Its function is to provide the common instance variables and methods you'll need for all employees.

In general, you should define a class (or a subclass of your root class) for each concept your application deals with.

When should I create a subclass?

Create a subclass whenever you need to change the behavior of a method or when you need to add more instance variables. If you write a method that does different things depending on the value of some instance variable, it's probably time to create a subclass.

What should be a method?

Typically, any action an object can make should be a method. It's considered good practice to provide methods to set and get the values of instance variables rather than allowing other functions to access them directly. If you find yourself passing an object as a parameter to a function to manipulate its instance variables, you should turn the function into a method.

When should I use procedural programming?

Object-oriented programming isn't the answer to all programming problems. Some problems are solved best by "old fashioned" procedural programming. In particular, procedural programming is better for procedures that are highly algorithmic or computationally intensive.

Where to Go Next

The next chapter describes how THINK C implements object-oriented programming. THINK C uses the syntax and many of the concepts defined in the C++ language. That chapter is fairly long and complex.

If you're not familiar with object-oriented programming, don't read all of Chapter 4. Instead, read up to the section called "Overview" on page 28, then try the "LearnOOP" tutorial in Chapter 5.

Chapter 7 is about the THINK Class Library, a collection of classes you can use to implement Macintosh applications.

Chapters 9–70 describe the main classes of the THINK Class Library in detail. Though these chapters are meant for reference, you might want to leaf through some of the class descriptions to get a sense of how the THINK Class Library is put together.

Using Objects in THINK C 4

This chapter shows you how to use the object-oriented extensions to THINK C. The THINK C implementation of objects is derived from C++ as described in *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup.

How to read this chapter

This is a long chapter, and there is a lot of information. Don't feel that you have to understand it all at once. Read a little bit at a time. Try out the examples. The Object Bullseye program in your THINK C package demonstrates many of the new features in a fairly small program.

Contents

Introduction 27
- What you should know 27
- A note on terminology 27

Using the Object Extensions 28

Overview 28

Declaring a Class 30
- Declaring root classes 30
- Declaring subclasses 31

Declaring and Using Objects 32
- Creating and deleting objects 32
- Assignment and objects 33
- Objects and memory 34

Accessing Members of a Class 35

Declaring Members of a Class 36
- Declaring instance variables 37
- Declaring class variables 38
- Declaring methods 39
- Virtual and non-virtual methods 39
- Class methods 40
- Examples of methods 40
- Declaring circular and forward references 41

Defining and Using Methods 42

Access Control 43
Using access specifiers 44
Access specifiers in subclass declarations 45
Protected access 46

Constructors and Destructors 48
Declaring and defining constructors 48
How constructors are invoked 49
Using access specifiers with constructors 50
Destructors 50

Allocators and Deallocators 51
Allocators 52
Deallocators 53
Default allocator and deallocator 54

Tips and Techniques 54
Organizing your classes 55
Objects and the Macintosh 55
Direct and indirect classes 58

Object Extension Options 58
Turning on the object extensions 58
Classes are indirect by default 59
Methods are virtual by default 59
Optimize monomorphic methods 59

Special Functions 59

Differences between THINK C and C++ 60
General differences 60
Keywords 61
Value objects and reference objects 61
THINK C extensions 62
Access declarations, friends, and multiple inheritance 62
Constructors and destructors 62
Scope resolution operator 62
Sizeof operator not allowed 62

Introduction

This chapter describes how THINK C implements objects. The THINK C 4.0 implementation of objects was based largely on Object Pascal, and borrowed its syntax from C++. This new implementation of objects builds on that foundation and adopts more the features of C++ like access control, constructors and destructors, and the `class` keyword.

This chapter begins with a section that tells you how to turn on the object extensions to THINK C and what libraries and headers you'll need. Next, you'll find a quick overview that tells you how to define a class, how to create an object, how to define a method, and so on. The sections that follow go into more detail on each aspect of the object extensions.

What you should know

Before you read this chapter, you should know the basics of object-oriented programming. Particularly, you should know about classes, instances, instance variables, and methods. If you need to learn about these things, look at Chapter 3, "Object-Oriented Programming." You should also be familiar with the Macintosh Memory Manager. To learn about it, see *Inside Macintosh II*, Chapter 1, "The Memory Manager."

A note on terminology

Though the implementation of objects in THINK C is based on C++, most of the terminology comes from Smalltalk and Object Pascal. If you read the introduction to object-oriented programming earlier in this manual, or if you've used objects in THINK C before, the terminology in this chapter will be familiar. If you know C++, the examples will look familiar, but the terminology might be foreign to you. Here's a short glossary to clear things up.

THINK C Term	C++ Term
method	member function, function member
instance variable	data member
superclass	base class
subclass	derived class
class method	static member function
class variable	static data member

Note

This chapter uses the word member to mean "instance variable or method."

Using the Object Extensions

See "Object Extension Options" on page 58 for information about options that affect the object extensions.

To use the object-oriented programming extensions to THINK C, you must make sure that you select the "THINK C + Objects" option in the Language Settings page of the **Options...** dialog.

Every program that uses objects in THINK C needs one of the libraries in the `oops Libraries` folder. The particular library you need depends on the kind of application you're using.

If you're writing...	you need this library
an application	`oops, oopsDebug`
an application with Far Code	`oopsFar`
an application with Far Data	`oopsFar`
a code resource	`oopsA4`
a desk accessory	`oopsA4`
a driver	`oopsA4`

The library called `oopsDebug` is designed to catch certain common errors that might occur while you're developing object-oriented applications. This library only works with applications that don't use Far Code or Far Data. If you want, you can make a copy of `oopsDebug` and rebuild it with the Far Code and Far Data options on.

For a description of these functions see page 59.

Any source file that uses the special functions `member()`, `new_by_name()`, or `class_name()` must include `<oops.h>`.

Overview

This section is a quick introduction to using the object extensions in THINK C. It gives you a broad picture of each step involved: declaring a class, creating an object, referring to instance variables and methods, and defining the methods of a class. The purpose of this section is to give you a working vocabulary. The sections that follow will build on what you learn here.

To use objects, first you need to declare a class. A class declaration describes the properties of a class and names the instance variables and methods. A class declaration is very much like a `struct` declaration.

```
struct MyClass : indirect {
  int  a;            // an instance variable
  int  b;            // an instance variable
  void M(int);       // a method declaration
  int  N(int, int);  // a method declaration
};
```


To use an object, you declare an object reference—a pointer to an object—and then use the new operator to allocate space for it. This is how you declare an object reference:

```
MyClass *anObject;    // ok: anObject is an
                      // object reference
```

But you can't do it like this:

```
MyClass anObject;     // error: illegal use of
                      // class type
```

Since `anObject` is a pointer, you need to allocate space in memory before you can use it. For objects, you use `new` like this:

```
anObject = new MyClass;
```

Some books on object-oriented programming refer to objects as "instances" of a class and to creation of objects as "instantiation."

The `new` operator creates a new object in your program's heap. In THINK C, most objects are allocated as handles, though you can create objects as pointers.

Now you can use the members—the instance variables and methods—of the object. To use an instance variable, you use the same syntax that you use to refer to the field of a `struct`. This example shows how to change the values of the instance variables `a` and `b`.

```
anObject->a = 10;
anObject->b = anObject->a++;
```

You use a similar syntax to call a method. This example sends a `M()` message to the object `anObject`:

```
anObject->M(6);
```

Of course, if you don't define the method, you'll get a link error when you try to run your program. Defining a method is almost like defining a function. To define a method `M()` for a class `C`, you name the function `C::M`. So the definition of the method `M()` in the example above might look like this:

```
void MyClass::M(int arg)
{
   int  t;
   int  b;

   t = N(arg, a);     // N is a method
                      // a is an instance
```

```
                        // variable
    b = t * 2;          // b is the local variable
    this->b = t * 2;    // this->b is the instance
                        // variable
}
```

Some special rules apply in a method definition. Whenever you define a method, it's as if this variable definition appeared at the beginning of the method:

```
register MyClass *this;
```

`This` is the current object—the object that received the message. When you use the name of a member in a method, it is as if you had written `this->` before the name. In the example above, the call to `N()` is the same as sending the message `N()` to `this`. If there's a local variable with the same name as an instance variable, you need to use `this->` explicitly to refer to the instance variable. In the example above, b is a local variable, so to refer to the instance variable you need to write `this->b`.

The next sections go into detail of each aspect of using objects: class declaration, method and instance variable declaration, class creation, method definition, and inheritance. Other sections cover topics that this quick introduction didn't even touch on: access control, constructors and destructors, and allocators and deallocators.

Declaring a Class

A class declaration describes the representation of objects and the operations they can perform. You can use the `class` keyword or the `struct` keyword to declare a class. When you declare a class, the name of the class becomes a type.

Declaring root classes

A **root class** is a class that does not inherit any members from any other class. The root class specifies the representation for a family of classes and whether references to members are made through a pointer or through a handle.

This is what root class declarations look like:

```
class class-name {
    member-declarations
};

class class-name : storage-specifier {
    member-declarations
};

struct class-name : storage-specifier {
    member-declarations
};
```

See "Direct and indirect classes" on page 58 for more information.

The storage specifier is either `direct` or `indirect`. If the storage specifier is `direct`, THINK C assumes that objects of the declared class are allocated as pointers and only does a single dereference to access members of those objects. If the storage specifier is `indirect`, THINK C assumes that they're allocated as handles and does a double dereference to access members.

See "Object Extension Options" on page 58 for information about options that affect the object extensions.

For classes declared with the `class` keyword, the storage specifier is optional. If the "Classes are indirect by default" option is on, objects of such a class are `indirect`; otherwise they're `direct` objects.

The member declarations describe the instance variables and methods that objects of the class will have.

Access specifiers are described in the section "Access Control" on page 43

When you use the `class` keyword, all the members of the class—all the instance variables and methods—are implicitly `private`. When you use the `struct` keyword, all the members of the class are implicitly `public`.

Declaring subclasses

A **subclass** is a class that inherits members from a root class or from another subclass. This is what a subclass declaration looks like:

```
class class-name : access-specifier superclass {
    member-declarations
};

struct class-name : access-specifier superclass {
    member-declarations
};
```

The access specifier determines how the members of the superclass are treated in the subclass. If the access specifier is `public`, the `public` and `protected` members of the superclass are treated as `public` and `protected` in the subclass. If the access specifier is `private`, the `public` and

For details and examples of access specifiers to superclasses, look at "Access specifiers in subclass declarations" on page 45.

`protected` members of the superclass are treated as `private` members of the subclass.

If the access specifier is missing, it defaults to `private` for classes declared `class` and `public` for classes declared `struct`.

Declaring and Using Objects

Unlike C++, THINK C does not allow you to declare objects that are not pointers.

Once you've defined a class, you can use its name to declare or define an **object reference**. An object reference is always declared as a pointer to a value of class type. This is how you declare an object:

class-name `*` *object-reference;*

Warning

Objects are always declared as pointers regardless of whether the storage specifier was `direct` or `indirect`. If you leave out the `*`, you will get the error "illegal use of class type."

An object reference definition allocates memory for the object reference only. It does not allocate memory for the object. To allocate the memory for the object, you must use the `new` operator described later.

Creating and deleting objects

Before you can use an object, you must allocate space for it in the heap. When you're finished with an object, you should delete it to reclaim its space. The operators `new` and `delete` create and delete objects.

To create an object, use `new` like this:

object-reference `= new` *class-name;*

To delete an existing object, use `delete` like this:

`delete` *object-reference;*

Note

In THINK C 4.0, `new` and `delete` were functions. In THINK C 5.0 `new` and `delete` are keywords supported by the compiler when the "THINK C + Objects" option is on.

Here is an example:

```
struct A : indirect {
    member declarations
};

struct B : A {
    member declarations
};

void f()
{
    A  *a;
    B  *b;

    b = new B;
    a = new B;

    delete a;
    delete b;
}
```

Example 4-1 Creating and deleting objects

Virtual methods are described in "Defining and Using Methods" on page 42.

In Example 4-1, note that `a` is declared as being of type `A`, but created as an object of class `B`. This means that `a` has a declared type of `A`, but a **run time** type of `B`. When you send a message to `a`, and the message refers to a virtual method, it is treated as if it were of class `B`.

Assignment and objects

You can assign one object reference to another if both objects are of the same class or if you're assigning an object of a subclass to an object of a superclass and the subclass was publicly derived from the superclass. The rules for object assignment are also used to match objects against the arguments of functions.

Here are some examples:

```
class A : indirect {
    member declarations
};

class B : public A {
    member declarations
};

class C : public B {
    member declarations
};
```

```
class K : A {
    member declarations
};

class L : K {
    member declarations
};

void g(A *x) { ... }

void f(void)
{
    A *a;
    B *b;
    C *c;
    K *k;
    L *l;

    a = c;    // OK: C is public subclass of A
    a = k;    // ERROR: K is private subclass of A

    g(b);     // OK: B is public subclass of A
    g(l);     // ERROR: L is private subclass of A

    a = new B;  // OK: B is public subclass of A
    a = new K;  // ERROR: K private subclass of A
}
```

Example 4-2 Legal and illegal object assignments

Objects and memory

See "Object Extension Options" on page 58 for information about options that affect the object extensions.

If a class is declared `indirect` (or if the "Classes are indirect by default" option is on and the class is declared without a storage specifier), THINK C allocates an object of that class as a handle. If a class is declared `direct`, an object of that class is allocated as a pointer. In both cases, object references are declared as pointers. If the space is allocated as a handle, THINK C does the additional dereference implicitly.

THINK C uses the Toolbox routines `NewHandle` and `NewPtr` to allocate space for objects, so you can use Memory Manager routines that work on pointers and handles. For instance, if you have an object of an `indirect` class, you might want to lock it down while you call other Toolbox routines that move memory.

You can define allocators and deallocators for a class if you want to allocate memory a different way. See "Allocators and Deallocators" on page 51 for more information.

Accessing Members of a Class

If you have an object reference, and you want to access one of its class members, use the same syntax that you use to refer to a member of a `struct` through a pointer. The form for accessing a member of a class is:

object-reference->member-name

The object reference is an object created with `new`. The member name is the name of the member that you want to access. If the member is not in the same class as the object reference, THINK C looks in the superclass.

The exact syntax for declaring members of a class is given in the next section. Many of the examples in this chapter access members of classes, so you should be familiar with the basic form. Come back to this section later to learn the details.

In Example 4-3, assume that class `B` is a descendant of class `A`—in other words, `A` is `B`'s superclass. Assume that `m` is an instance variable declared in `B` and that `n` is an instance variable declared in `A`.

```
struct A : indirect {
    short n;
    other member declarations
};

struct B : A {
    short m;
    other member declarations
};

void g(void)
{
    B  *b;

    b = new B;          // b is an object of B
    b->m = 1;           // m is a member of B
    b->n = 1;           // n is a member of A
                        // inherited by B
    delete b;
}
```

Example 4-3 Accessing members of a class

If the member you want to access is a method, use the same syntax, but as if you were calling a function.

```
void g(void)
{
   B  *b;

   b = new B;          // b is an object of B
   b->Meth();          // Meth is a method of B
                       // or inherited by B
}
```

Example 4-4 Calling a method

When you access an instance variable or call a method, THINK C starts looking for the member in the run time class of the object reference. You can use the **scope resolution operator** : : (two colons) to specify in which class the search should begin. The more general form of member access is:

object-reference->class-name: : *member-name*

Virtual methods are described in "Defining and Using Methods" on page 42.

For example, suppose that B and A both declare a virtual method M() and that A is a superclass of B. You can use the scope resolution operator to bypass the inheritance rules like this:

```
void g(void)
{
   B  *b = new B;
   A  *a;

   a = new B;          // a's run time class
                       // is B

   b->M();             // call B's M()
   a->M();             // since M() is virtual,
                       //    calls B's M()
   a->A::M();          // calls A's M()
}
```

Example 4-5 Using the scope resolution operator

Declaring Members of a Class

To learn about access specifiers see "Access Control" on page 43.

In THINK C, members of a class must be either data or functions. A class can declare any number of instance variables and must declare at least one method. Access specifiers may be placed wherever they're needed within the member declarations.

Declaring instance variables

The declaration of an instance variable in a class is similar to the declaration of a field in a struct.

```
struct A : indirect {
   int   a;
   int   b;
   at least one method declaration
};

class B : public A {
   int   c;
   int   d;
   at least one method declaration
};
```

Example 4-6 Declaring instance variables.

A subclass inherits all the instance variables of the superclass as if they had been declared in the subclass. In Example 4-6, class B inherits the instance variables a and b from class A. It is as if class B had been declared like this:

```
struct B : indirect {
   int   a;
   int   b;
   int   c;
   int   d;
   at least one method declaration
};
```

Example 4-7 The effect of inheritance

If you declare an instance variable with the same name as an instance variable in a superclass, two separate instance variables exist. An object of the subclass has two different instance variables that have the same name.

```
struct A : indirect {
   int   a;
   int   b;
   at least one method declaration
};

struct B : A {
   long  b;
   int   m;
   at least one method declaration
};
```

Example 4-8 Instance variables with the same name

In Example 4-8 the class A has an instance variable named b. Class B, a descendant of class A, has another instance variable named b. If you create an object of class B and access the instance variable b, you'd be referring to the b defined in class B. You can use the scope resolution operator to refer to the instance variable in the superclass.

```
void F (A *p, B *q)
{
   p->b = 10;      // This is A::b
   q->b = 10;      // This is B::b
   q->A::b = 10;   // This is A::b
}
```

Example 4-9 Referring to a specific instance variable

Declaring class variables

An instance variable declared `static` is a **class variable**. A class variable isn't part of an object of its class. Instead, all the objects of the class share the class variable. You might want to use a class variable to keep information that pertains to all the objects of the class. For instance, if you have a class that represents some limited resource, you can use a class variable to keep track of its use.

Before you can use a class variable you need to define it. Its declaration in the class declaration does not allocate space for it.

```
struct A : indirect {
   static int count;     // class variable

   int   a;
   int   b;

   void  F(void);
};

int A::count = 0; // defining and initializing
                  // the class variable
```

Example 4-10 Declaring and defining a class variable

Unlike other members of a class, you don't need an object reference to refer to a class variable. If you do provide an object reference, it is used to determine the class. So the statements below all refer to the same class variable:

```
void g(A *a)
{
   A::count = 10;    // set count to 10
   a->count = 20;    // same as A::count = 20
}
```

Example 4-11 Accessing a class variable

To learn about access control, see "Access Control" on page 43.

The access control rules apply to class variables. If the class variable `count` in Example 4-10 had been declared `private`, only methods of class `A` could access and modify it. The only exception to the access control rules is that you are always allowed to initialize a class variable in global scope.

Declaring methods

See "Object Extension Options" on page 58 for information about options that affect the object extensions.

The syntax for declaring a method is the same as for declaring a function. You can declare a method normally, or you can declare a method to be `virtual` or `static`. If the "Methods are virtual by default" option is on, all non-static method declarations are implicitly `virtual`.

```
struct A : indirect {
   int   a;
   int   M(int);          // normal declaration
   virtual int V(int);    // virtual declaration
   static int S(int);     // static declaration
};
```

Example 4-12 Method declarations in a class

In THINK C, every class—whether declared `struct` or `class`—must have at least one method.

A method declared in a subclass can have the same name as a method declared in its superclass. If a method is declared `virtual` in a superclass, a method of the same name in a subclass is also `virtual` whether or not the method in the subclass is declared `virtual`. The return types and argument lists for virtual methods must match between superclass and subclass.

Virtual and non-virtual methods

Virtual methods are the mechanism that makes object-oriented programming work. When you send an object a message—that is, invoke one of its methods—the actual method that executes depends on the run-time class of

the object. When you declare a method in a subclass for a virtual method that exists in the superclass, you are overriding the method.

If a method in a superclass is declared non-virtual (and the "Methods are virtual by default" option is off), you can declare a method with the same name in a subclass. The two methods may have different return types and argument lists. The non-virtual methods in a subclass does not override a method in the superclass. There are simply two methods with the same name.

Class methods

A method declared `static` is a **class method**. A class method doesn't behave like a non-virtual or virtual method. Instead, it's a function whose scope is limited to the class that declares it and to the subclasses that have access to it. Class methods are useful as auxiliary routines. In Macintosh programs, you can use a class method declared `pascal` as a callback routine.

Note

You cannot take the address of a virtual or non-virtual method, but you can take the address of a class method.

Examples of methods

The following example illustrates the different method declarations and method invocations.

The use of `virtual` here is optional, but it is good programming style.

```
struct A : indirect {
   virtual  void V1(void);
   virtual  void V2(void);
            void M1(void);
   static   void S1(void);
};

struct B : A {
   virtual  void V1(void);
            void M1(int);
};

void g(void)
{
   A  *a;          // declared type is A *
   B  *b;

   a = new B;      // run-time class of a is B
   b = new B;

   a->V1();        // invokes B::V1()
   b->V1();        // invokes B::V1()
   b->V2();        // invokes A::V2()
   a->M1();        // invokes A::M1()
   b->M1(1);       // invokes B::M1()
   b->A::M1();     // invokes A::M1()
   b->S1();        // invokes A::S1()
   A::S1();        // invokes A::S1()
}
```

Example 4-13 Method declarations and invocations

When you call a virtual method, the THINK C message dispatcher looks at the run-time class of the object and invokes the appropriate method. In Example 4-13 above, the run-time class of both `a` and `b` is class `B`, so when `V1()` is called, class B's `V1()` is invoked.

If the method is monomorphic and the "Optimize monomorphic methods" option is on, the linker bypasses the dispatcher and generates a call directly to the method. A monomorphic method is a method that does not override another method and is never overridden.

Declaring circular and forward references

Since the name of a class is known by the time you declare members of a class, you can use its name for circular references. For example:

```
struct AList : indirect {
   int   someInfo;
   AList *nextItem;
   method-declarations
};
```

Example 4-14 A circular reference.

If a class declaration needs to refer to another class that hasn't been declared yet, you can use the `class` keyword to name the class before you use it:

```
class B;       // name the class before using it

struct A : indirect {
   int   a;
   void  F(B *b);
};

struct B : indirect {
   int   m;
   void  G(A *a);
};
```

```
g (A *p, B *q)
{
   p->F(q);     // now it works
}
```

Example 4-15 A forward reference

Use the same technique for declaring classes that are mutually dependent:

```
class B;        // name the class before using it

struct A : indirect {
   B     *anObject;
   void  F(void);
};

struct B : indirect {
   A     *anObject2;     // A is already declared
   void  G(void);
};
```

Example 4-16 Mutually dependent classes

Defining and Using Methods

You define a method the same way you define a function, except that you include the class name before the method name like this:

return-type *class-name*`::`*method*`(`*parameter-decls*`)`
`{`
    function-body
`}`

Within a method, the compiler supplies an implicit declaration so you can refer to the object that received the message:

```
register class-name *this;
```

The symbol this is analogous to the implicit variable `self` *in Object Pascal or Smalltalk.*

The symbol `this` refers to the object receiving the message. Within a method definition, you can omit `this->` when you refer to an instance variable or when you call a method in the same class or superclass as `this`.

If your method definition redeclares a variable with the same name as an instance variable or method, you can use `this->` or the scope resolution operator to recover the original meaning.


```
struct A : indirect {
   int   a;
   int   b;
   void  M(void);
};

void A::M(void)

{
   int   a;     // local variable with same
                // name as an instance variable

   b = 7;          // sets instance variable to 7
   a = 5;          // sets local variable to 5
   this->a = a;    // sets instance variable to 5
   A::a = a;       // sets instance variable to 5
}
```

Example 4-17 Local and instance variables with the same name

Within a method definition, you may want to use the superclass's method. For example, to add functionality to a method, you use the superclass's method, then add more code to augment the behavior.

This is how you call a superclass's method:

superclass`::`*method*`(`*args*`)`

To refer to the immediate superclass of the current class, you can use the word `inherited`, like this:

`inherited::`*method*`(`*args*`)`

When you use the scope resolution operator to invoke a method, THINK C bypasses the message dispatcher and generates a direct call.

Note

This use of the word `inherited` is a THINK C extension and not found in C++.

Access Control

THINK C implements an access control mechanism based on C++. Access control lets you specify which instance variables and methods are available to subclasses and to other classes. You can use the access control mechanism to make sure that only subclasses change the values of instance vari-

ables or to make sure that certain methods are available only to the class that defines them.

An instance variable or method can be `public`, `private`, or `protected`.

Access Control Labels	Meaning
`private`	The instance variable or method is available only to methods declared in the class that the declaration appears in.
`protected`	The instance variable or method is available only to methods declared in the class that the declaration appears in or in subclasses of that class.
`public`	The method or instance variable is available anywhere.

Note

The words `public`, `private`, and `protected` are not keywords in THINK C as they are in C++. THINK C interprets the access specifiers only in class declarations.

The way you declare a class—whether you use the `class` keyword or the `struct` keyword—affects the default access control for that class.

Classes declared with...	have this access control
`class`	`private`
`struct`	`public`

Using access specifiers

An access specifier—`public`, `private`, `protected`—can appear before any instance variable or method declaration in a class declaration. The access specifier is in effect until the next specifier or until the end of the class declaration.

Here are some examples:

```
class ClassA {
   int   a;          // a is private by default
private:
   int   b;          // b & c are private
   int   c;
protected:
   int   d;          // d is protected
public:
   int   E(void);    // E() is a public method
private:
   int   F(int x);   // F() & g are private
   int   g;
};
```

Example 4-18 Access specifiers for members of a class

Remember that when you use the `class` keyword to declare a class, the members of that class are `private` by default. If you use `struct` to declare a class, all the methods are public by default:

```
struct ClassB : indirect {
   int   h;          // h is public
   int   M(int x);   // so is M()
private:
   int   i;          // but i and j are private
   char  j;
};
```

In these examples, the instance variables `ClassA::a`, `ClassA::b`, `ClassA::c`, `ClassA::g`, and the method `ClassA::F()` are all private. The only places that you'd be allowed to refer to them are in the methods `ClassA::E()` and `ClassA::F()`. Since `E()` is the only public method of `ClassA`, that's the only message you can send to an object of `ClassA`.

Access specifiers in subclass declarations

When you create a new class, you can use the `public` and `private` access specifiers to control the access to the instance variables and methods of the superclass.

If you specify that the superclass should be `public`, then all the `public` and `protected` members of the superclass will be `public` and `protected` in the subclass. If you specify that the superclass should be `private`, the `public` and `protected` members of the superclass will be private members of the subclass.

If you don't provide an access control specifier, THINK C assumes `private` when the subclass is declared with `class` and `public` when the subclass is declared with `struct`.

```
class A {              // a root class
   int   a;
public:
   int   b;
   void F(void);
};

class B1 : public A {
   int   m;
public:
   int   n;
   void  G(void);
};

class B2 : private A {
   int   w;
public:
   int   x;
   void  Q(void);
};

void fn(void)
{
   B1    *p = new B1;
   B2    *q = new B2;

   p->a;        // violation: A::a is private
   p->b;        // ok: A::b is public to B1
   q->b;        // violation: A is private to B1
}
```

Example 4-19 Access specifiers for superclasses

Protected access

Protected access lets subclasses access members of a superclass without making those members public. For instance, in an abstract class, you might want to make certain implementation-level details available only to subclasses of that class, but not to other classes.

The rules for protected access are not as intuitive as the ones for private and public access. Access to a protected member is restricted not only by the class of the object reference but also by the class of the method the reference appears in.

To refer to a protected member, three conditions must be true:

1. The reference must appear in a method definition.
2. The method in which the reference appears must be of the same class as the class in which the protected member is declared (or in a subclass of that class).
3. The object through which the reference is made must be in the same class as the method in which the reference appears (or in a subclass of that class).

The examples below assume these declarations:

```
struct A : indirect {
protected:
   int   a;
   void  M1(void);
};

struct B1 : A {
   void  M2(void);
};

struct B2 : A {
   void  M3(B1 *q);
};
```

In Example 4-20, the method that the protected member `a` is referenced in is of the same class as `a`. Since the reference is through `this->`, which is of the same class as `M1()`, the instance variable `a` is accessible.

```
void A::M1(void)
{
   a = 1;    // ok
}
```

Example 4-20 Protected access through the same class

In Example 4-21 the method that the reference appears in is defined in a subclass of the class that `a` is declared in. In other words, `B1` is a subclass of `A`. As in the previous case, the reference is through `this->`. Since `this` is of the same class as the method, `B1`, the instance variable `a` is accessible.

```
void B1::M2(void)
{
   a = 1;    // ok
}
```

Example 4-21 Protected access through a subclass

In Example 4-22 M3() is of class B2 which is a subclass of A, the class the protected member is declared in. The reference, though, is through q, which is of class B1. Since q is not the same class as the method, B2, the protected instance variable is not accessible.

```
void B2::M3(B1 *q)
{
   q->a = 1;    // illegal
}
```

Example 4-22 Illegal protected access

Constructors and Destructors

A class can have two special methods that are called when an object is created and destroyed. The method that's called when an object is created is called a **constructor**, and the method called when an object is deleted is called a **destructor**.

Declaring and defining constructors

In THINK C, you can't call a constructor directly to create an object as you can in C++.

A constructor is a method that has the same name as the class it's declared in. For example, the constructor for the class SomeClass is named SomeClass::SomeClass(). A constructor is invoked automatically when you create an object with new. If an object has a superclass, the superclass's constructor is called before the current class's constructor is called.

There are some special rules for declaring a constructor. You cannot declare a constructor virtual or static. You cannot specify a return type for a constructor—not even void. This is how you declare a constructor:

```
class SomeClass {
   int            theLastCount;
   AnotherClass   *list;
   SomeClass(void);
   virtual int GetCount();
   ...
};
```

Example 4-23 Declaring a constructor.

The same rules apply when you define the constructor.


```
SomeClass::SomeClass (void)
{
   theLastCount = GetCount;

   list = new AnotherClass;
}
```

Example 4-24 Defining a constructor

How constructors are invoked

If a class is not a root class, the superclass's constructors are called from the root class to the class of the object being created. For instance, suppose you have these classes:

```
class A {
public:
   A(void);
   virtual void F(void);
};

class B : A {
public:
   B(void);
   virtual void F(void);
};

class C : B {
public:
   C(void);
   virtual void F(void);
};


void f(void)
{
   C  *theC;

   theC = new C;
}
```

Example 4-25 Constructors in a class hierarchy

When you create an object of class C the constructors are called in order: A(), B(), and C() just the way you'd expect. One thing you might not expect is that while A() is running, the class of the current object this is A, while B() is running this is of class B, and it's not until C() executes that this is of class C. The implication of this behavior is that if your construc-

tors call the method `F()`, a different `F()` is invoked. In `A()`, `A::F()` will execute; in `B()`, `B::F()` executes; and in `C`, `C::F()` executes.

Using access specifiers with constructors

You can use access specifiers to control access to constructors and destructors. If you declare the constructor of class A protected, only subclasses of class A are allowed to create those objects. If you declare a constructor private, the only way you can create an object of that class is to do so in a class function. That's why the constructors in Example 4-25 are declared `public`.

Destructors

A destructor is the opposite of a constructor. When you delete an object, its destructor is invoked. If the object has a superclass, its destructor is invoked, and so on until the destructor of the root class is reached.

Similar rules about declaring constructors apply to destructors. A destructor cannot be `static`, and a destructor cannot have a return type. In C++ a destructor can be declared `virtual`. In THINK C, a destructor is *always* `virtual`. Here's how you declare a destructor:

```
class SomeClass {
   int            theLastCount;
   AnotherClass   *list;
   SomeClass(void);          // Constructor
   ~SomeClass(void);         // Destructor
   virtual int GetCount();
   ...
};
```

Example 4-26 Declaring a destructor

And this is how you define a destructor:

```
SomeClass::~SomeClass(void)
{
   delete list;
}
```

Example 4-27 Defining a destructor

The destructor invoked depends on the run time type of the object being deleted. Look at this example:

```
class A {
public:
   ~A(void);
};

class B : A {
public:
   ~B(void);
};

class C : B {
public:
   ~C(void);
   virtual F()
}

f ()
{
   A  *theThing = new C;
   ...
   delete theThing;
}
```

Example 4-28 Destructors in a class hierarchy

In this example, `theThing` is declared as being of class `A`, but its runtime type is class `C`. When `theThing` is deleted, the destructor invoked is `~C` not `~A`. The destructors are called in this order: `~C()`, `~B()`, `~A()`. As destructors are invoked, the type of the object changes as it's destroyed. In `~C()`, the class of `this` is `C`, in `~B()` it's `B`, and in `~A()` it's `A`.

Warning

This behavior matches the C++ behavior for virtual destructors. For maximum compatibility with C++, be sure to declare destructors `virtual`.

Allocators and Deallocators

Allocators and deallocators let you customize how memory is allocated for an object. An allocator allocates memory for the object, and a deallocator returns an object's memory to the free store. Allocators execute before any constructor, and deallocators execute after the last destructor.

Allocators and deallocators are always `static` and have `public` access regardless of how they're actually declared.

Allocators

Allocators allocate memory for an object. If C is the name of a class, the name of its allocator is `C::operator new()`. The function `operator new` takes the number of bytes to allocate as a parameter and returns a pointer to the allocated memory:

The type `size_t` is defined in `<stddef.h>`. THINK C lets you use `long` or `unsigned long` instead of `size_t`.

```
class C {
   void * operator new (size_t);
   ...
};
```

Example 4-29 Declaring an allocator

This is what the allocator might look like:

```
void * C::operator new (size_t nBytes)
{
   Handle   h;
   h = NewHandle(nBytes);
   if (h == NULL) {
      // try to get memory from a secret stash
   }
   return h;
}
```

Example 4-30 Defining an allocator

Since allocators execute before any constructor, you can't do anything with the instance variables or methods of the object. All you can do is allocate memory.

Warning

Be sure that your allocator allocates the right kind of memory. If your class is declared `indirect`, your allocator should allocate handles. If your class is declared `direct`, it should allocate pointers.

If a class and its superclass have allocators, the allocator called is for the class being created.

```
class A {
   void * operator new (size_t);
   ...
};

class B : public a{
   void * operator new (size_t);
   ...
};

f()
{
   A  *p;
   A  *q;

   p = new A;      // uses A::operator new()
   q = new B;      // uses B::operator new()
}
```

Example 4-31 Only the allocator for the class being created executes

Deallocators

Deallocators deallocate memory for an existing object. If C is the name of a class, the name of its deallocator is `C::operator delete()`. The function `operator delete` takes a pointer to an object as a parameter and returns nothing:

```
class C {
   void operator delete (void *);
   ...
};
```

Example 4-32 Declaring a deallocator

The deallocator might look like this:

```
void C::operator delete (void *theObject)
{
   if (IsPrivateStore(theObject)) {
      // return to private store
   } else
      DisposHandle ((Handle) theObject);
}
```

Example 4-33 Defining a deallocator

Since deallocators execute after destructors, the object that's being deallocated no longer exists. The only thing you can do in a deallocator is return the object's memory to the free store.

If a class and its superclass have deallocators, the allocator called is for the runtime class of the object being deleted.

```
class A {
   void operator delete (void *p);
   ...
};

class B : public a{
   void operator delete (void *p);
   ...
};

f()
{
   A  *p = new A;
   A  *q = new B;
   ...
   delete p;      // uses A::operator delete()
   delete q;      // uses B::operator delete()
}
```

Example 4-34 Only the deallocator for the class being deleted executes

Default allocator and deallocator

If a class does not define an allocator or deallocator, THINK C uses its default built-in ones. If the class is declared `indirect`, or if the "Classes are indirect by default" option is on, the default allocator calls the Toolbox routine `NewHandleClear`. If the class is a direct class, the default allocator calls `NewPtrClear`. The default deallocator calls the Toolbox routine `DisposHandle` for indirect classes and `DisposPtr` for direct classes.

In THINK C you cannot change the default allocator or deallocator. Instead, you can specify an allocator and a deallocator for a root class and declare all classes to be descendants of that class.

Tips and Techniques

This section gives some general tips to make writing object-oriented programs a little easier. It describes how to organize your classes, how to initialize objects easily, how to make sure that your objects behave correctly under the Macintosh Memory Manager, and how the THINK C implementation of objects relates to C++.

Organizing your classes

It's a good idea to isolate classes to make them easy to use and re-use. For each class, create a `.h` file that contains the class declaration and a corresponding `.c` file that contains the method definitions. The names of the files should be the same as the names of the classes.

Your `.h` file should first include the `.h` file for the superclass of the class you're declaring. Next you declare your class. Since several classes may try to include your header file, you need a way to keep it from being included more than once. THINK C gives you a way to do this easily.

To learn more about once-only headers, see "Once-only headers" in Chapter 9 of the THINK C User Manual.

To prevent a header file from being included more than once, use the `#pragma once` preprocessor directive. This directive tells the compiler that the header file should only be included once for that source file.

Assume you're defining a class called `ClassName`. The file that implements your methods would be called `ClassName.c`. The corresponding header file would be called `ClassName.h`. Here's an example of what `ClassName.h` looks like:

```
#pragma once // To keep from #including it again

#include "SuperClass.h"

struct ClassName : SuperClass {
      ...
};
```

Example 4-35 Using #pragma once in a class declaration file

Objects and the Macintosh

See "Object Extension Options" on page 58 for information about options that affect the object extensions.

The most important thing to bear in mind about THINK C objects is that if they're declared `indirect`, or if the "Classes are indirect by default" option is on, the object is a Macintosh handle. Any reference to an instance variable through an indirect object is implicitly a handle reference. Because of this implementation, your program should not rely on the addresses of instance variables, particularly in calls to Toolbox routines that may move memory.

Take a look at this hypothetical implementation of a window object:

```
class Window : indirect {
   WindowPtr   theWindow;
   Rect        growRect;
   Rect        dragRect;

public:
      Window(void);  // constructor
      ~Window(void); // destructor

virtual  short    Hit(Point where);
virtual  void     Drag(Point where);
virtual  short    Draw(void);
};
```

This is the way you might implement the `Drag()` method:

```
void Window::Drag(Point where)
{
   DragWindow(theWindow, where, &dragRect);
}
```

The problem with this definition is that it will probably crash at some point. `DragWindow` is a Macintosh Toolbox routine that may move memory, including the handle `this`. By the time `DragWindow` is invoked, the address of `dragRect` has already been calculated. If `DragWindow` moves memory, and `this` moves to a different part of memory, the address being used for `dragRect` will actually refer to some other, possibly invalid, data.

A better definition is this:

```
void Window::Drag(Point where)
{
   Rect  dragRect;

   dragRect = this->dragRect;
   DragWindow(theWindow, where, &dragRect);
}
```

In this case, `Drag()` copies the instance variable into a local variable with the same name. Since the address of the local variable never changes within a function block, this use is safe. Though it's not necessary to name the local variable the same as the instance variable, this shadowing technique will keep you from accidentally using the address of an instance variable.

You should not rely on the order of evaluation in any assignment statement. To be completely safe, program defensively, and assume that any operation that may move memory, will move memory.

For instance, be careful about passing arrays as arguments to functions that move memory:

```
struct A : indirect {
   Handle theHandles[100];
   void M(void);
};

void MakeHandles(short howMany, Handle hArray[])
{
   short i;

   for (i = 0; i < howMany; i++)
      hArray[i] = NewHandle(1024);
}

void A::M(void)
{
   MakeHandles(100, theHandles);
}
```

Example 4-36 Moving memory unexpectedly

Whenever you pass an array as an argument to a function, it is passed by reference. That means that when `A::M()` calls `MakeHandles`, it gets the address of the first element of `theHandles`. When `MakeHandles` calls `NewHandle`, there's a good chance that the object will move, and at that point, the address that `MakeHandles` is using is no longer valid.

You can use the Memory Manager routines `HLock` and `HUnlock` and `HGetState` and `HSetState` to keep indirect objects from moving around. Most of the time, you won't need to use these routines if you use the shadowing technique shown above. In the case of arrays, though, your only recourse is to lock the object:

```
void A::M(void)
{
   SignedByte hState;

   hState = HGetState(this);
   MoveHHI(this)
   HLock(this)

   MakeHandles(100, theHandles);

   HSetState(this, hState);
}
```

Example 4-37 Locking an object to keep it from moving

Direct and indirect classes

The only difference between direct and indirect classes is the way objects are allocated in memory. Objects of a direct class are allocated as pointers, while objects of an indirect class are allocated as handles.

Since handles can move freely in memory, indirect objects are more "memory-friendly." On the other hand, referring to any member of an indirect object needs an extra level of indirection, so they're slightly slower. Direct objects only require one level of indirection, so they're faster. But direct objects might tend to fragment your heap.

Object Extension Options

This section describes the options that control the object extensions to THINK C. For additional information about the **Options...** dialog, see "Using the Options... Dialog" in Chapter 10 of the *THINK C User Manual*.

Turning on the object extensions

For the object extensions to work, you must set the "THINK C + Objects" option on in the Language Settings page of the **Options...** dialog. Turn on the "Language Extensions" check box, and select the "THINK C + Objects" radio button.

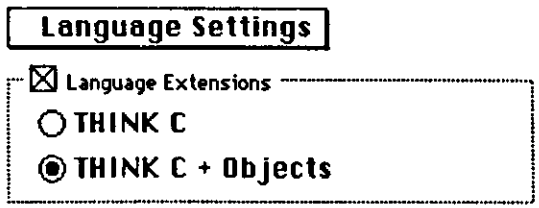

Figure 4-1 The THINK C + Objects option

Three options in the Compiler Settings page control how THINK C generates code for programs that use the object extensions:

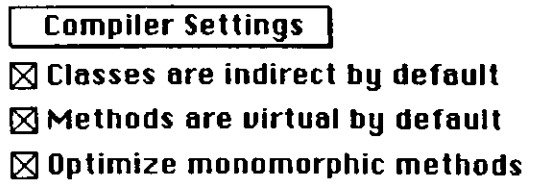

Figure 4-2 Compiler settings that affect object extensions

Classes are indirect by default

If this option is on, root classes that you declare with the `class` keyword, and for which you do not provide a storage specifier, are treated as if the storage specifier was `indirect`.

This option doesn't affect classes declared with the `struct` keyword because classes declared that way must have a storage specifier.

You can still use the storage specifiers `direct` and `indirect` in any root class declaration regardless of how this option is set. In the example below, class A is either `direct` or `indirect`, depending on how this option is set. Class B is always `direct` regardless of how the option is set.

```
class A {
    member declarations
};

class B : direct {
    member declarations
};
```

Example 4-38 Specifying class storage

Methods are virtual by default

If this option is on, every method in a class is implicitly virtual. For maximum compatibility with THINK C 4.0, you should leave this option on. For maximum compatibility with C++, you should turn this option off. If you plan to take advantage of non-virtual methods, be sure that this option is on.

Regardless of the way this option is set, remember that:

- Static methods are never virtual
- Allocators and deallocators are never virtual
- Constructors are never virtual
- Destructors are always virtual

Optimize monomorphic methods

When this options is on, THINK C optimizes calls to monomorphic methods—virtual methods that are never overridden—so the call does not have to go through the method look-up table. You should leave this option on.

Special Functions

This section describes functions that let you create objects by name and test for class membership. These functions are extensions to THINK C that do

not appear in C++. Be sure that you `#include <oops.h>` in the source file that uses these functions.

new_by_name

```
void *new_by_name (char *aName);
```

Create a new object. AName is a C string that names the class. If there is no class with the given name, new_by_name returns NULL.

The named class must be defined in your program. If your program defines a class but never creates an instance of it, the smart linker may remove the class definition, so you'll be unable to use new_by_name to create an instance of that class. One way to get around this restriction is to create a dummy object, then immediately delete it. Another way is to use the member() function to refer to the class.

class_name

```
char *class_name (void *anObject);
```

Return a string that is the name of the class the anObject belongs to. This function does not check anObject to make sure that it is a valid object reference.

member

```
char member (void *anObject, void *aClass);
```

Return 1 if anObject is an instance of aClass or an instance of one of its ancestors, zero otherwise.

Differences between THINK C and C++

The syntax that THINK C uses for objects is a subset of C++ as described in *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup (Addison-Wesley). THINK C implements simple objects without multiple inheritance. THINK C does not implement any non-object extensions from C++ such as operator or function overloading.

General differences

The main difference between THINK C and C++ is that, at its core, THINK C is an ANSI C compiler, and there are significant yet subtle differences between ANSI C and C++ besides the obvious ones. Keep in mind that C and C++ are completely different languages.

There are a few anomalous cases where something that works in THINK C might not work in C++, even if you've taken care not to use any of the THINK C extensions. Or there might be cases where a C++ program that restricts itself to the implementation of objects in THINK C will behave differently in THINK C.

For guidance in these anomalous cases, look at section 18.2 "C++ and ANSI C" of *The Annotated C++ Reference Manual*. The scope rules of ANSI C and C++ are significantly different. Refer to section 3.1.2.1 "Scopes of Identifiers" of the ANSI C Standard (*American National Standard for Information Systems—Programming Language—C* ANS X3.159-1989).

Keywords

When the "THINK C + Objects" language extension option is on, THINK C reserves the following names as keywords:

```
class       delete
new         virtual
```

The following names are interpreted specially in context, but they are not keywords:

```
direct      indirect      inherited
operator    private       protected
public      this
```

Value objects and reference objects

In THINK C you can refer to an object only through a pointer. C++ lets you write this:

```
class myClass {
   int   someField;
   void  SomeMeth(void);
} aBadObject;
```

Example 4-39 A value object

In THINK C, you have to write it like this:

```
class myClass {
   int   someField;
   void  SomeMeth(void);
} *aGoodObject;

aGoodObject = new myClass;
```

Example 4-40 A reference object

THINK C extensions

THINK C adds some features that do not appear in C++. If you use these features, your program will not port directly to C++.

- The word `inherited` is allowed as a scope qualifier to refer to a class' superclass.
- The `direct` and `indirect` words as storage specifiers for class declarations.
- The `new_by_name` function for instantiation by name.
- The `class_name` function to get a class name from an object.
- The `member` function to test object membership in a class.

Access declarations, friends, and multiple inheritance

THINK C does not support access declarations that allow classes to adjust access to specific members of a superclass .THINK C does not support friends or multiple inheritance.

Constructors and destructors

The only kind of constructor that THINK C supports is a default constructor—that is, a constructor with no arguments. The only way you can invoke a constructor is by creating an object with new. You can't create an object by calling its constructor directly as you can in C++.

In THINK C, all destructors are implicitly `virtual` whether or not declared `virtual` even if the "Methods are virtual by default" option is off.

Scope resolution operator

THINK C does not support the unary scope resolution operator.

Sizeof operator not allowed

You cannot use the `sizeof` operator to take the size of a class name or of an object. This use was allowed in THINK C 4.0.

Tutorial: LearnOOP 5

LearnOOP is a shape-drawing program that introduces you to the basics of object-oriented programming. Each of the shapes that LearnOOP draws is an object. You'll see how the routines that operate on objects are encapsulated in the class declaration. You'll learn how dynamic binding and how inheritance work. Then you'll define a new class yourself.

If you already know object-oriented programming, you don't have to read this chapter. Go on to the next chapter to learn about the class browser.

Before you begin

Make sure that you've installed THINK C on your Macintosh as described in the instructions in the *THINK C User Manual*, Chapter 2, "Installing THINK C.". You don't need to install the THINK Class Library to run this tutorial.

What you should know

This tutorial assumes that you know how THINK C works and how to use the THINK C Debugger . You should know how to open a project, how to run a project, and how to use the editor. If you need to learn how to do any of these things, see the tutorial in Chapter Chapter 3, "Tutorial: Hello World," of your *THINK C User Manual*. If you need to learn how to use the debugger, do the tutorial Chapter 5, "Tutorial: Bullseye," before you try LearnOOP.

Contents

What Does LearnOOP Do? 65
How Does LearnOOP Work? 67
How Do Objects Know How to Draw Themselves? 68
The LearnOOP Class Declarations. 71
How Do You Create an Object? 74
How Do You Define a New Class? 75
But How Does It Know? 77
Where to Go Next 78

What Does LearnOOP Do?

To begin this tutorial, double-click on the `LearnOOP.π` project in the `LearnOOP` folder. If you're already in THINK C, close any existing project, and choose `LearnOOP.π` from the standard file dialog. The project window looks like this:

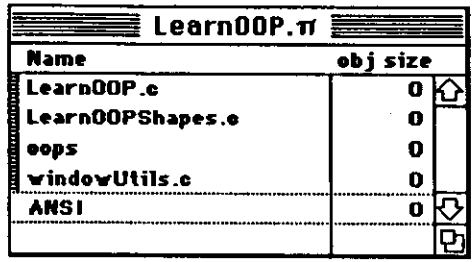

Figure 5-1 The LearnOOP project window

Choose **Run** from the **Project** menu to start the program. THINK C loads the libraries and compile the source files before LearnOOP starts running. It will take a minute or so.

The LearnOOP program uses the console window to prompt you for something to do, and it draws in another window called the Drawing window. When you run the program, LearnOOP asks you if you want to create an Oval or a Rectangle:

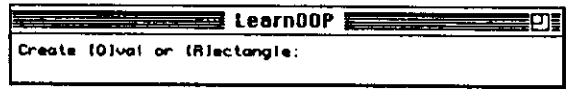

Figure 5-2 LearnOOP asking for a shape

If you want to draw an oval, type an `O`; if you want to draw a rectangle, type an `R`. To quit the program altogether, type a `Q`. Right now, type an `O`, and press the Return key.

After you tell LearnOOP what kind of shape to draw, it asks you to give it the top, left, bottom, and right coordinates of a rectangle that completely encloses the shape:

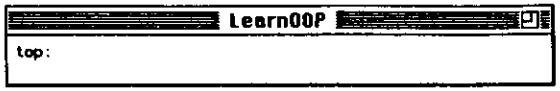

Figure 5-3 LearnOOP asking for a coordinate

The point 0,0 is at the top left corner of the window. To see how this works, try typing these values: `10, 10, 40, 90`. Press the Return key after each value. LearnOOP draws the shape in the drawing window like this:

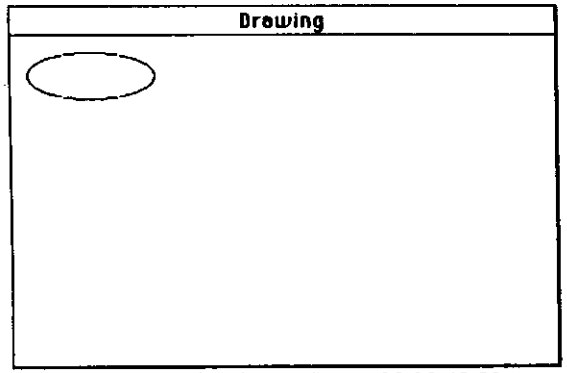

Figure 5-4 An oval with coordinates 10 10 40 90

After it draws the first shape, LearnOOP asks what kind of shape to draw next. Try a few more shapes before you go on to the next section.

How Does LearnOOP Work?

If LearnOOP is still asking you for a kind of shape, type a `Q` to make it stop.

Double-click on the file named `LearnOOP.c` in the project window. The program that draws shapes is pretty simple. Look at `main` at the end of the file:

```
/****
 * main
 *
 * Prompt for a shape and its location. Then
 * draw it in the drawing window.
 *
 ****/

main()
{
   CShape   *aShape;
   short    top, left, bottom, right;

   SetUpWindows();

   while (aShape = ChooseShape()) {
      top = GetNum("top");
      left = GetNum("left");
      bottom = GetNum("bottom");
      right = GetNum("right");

      aShape->SetBounds(top, left,
                        bottom, right);
      aShape->Draw();
   }
}
```

`SetUpWindows()` is in the file `windowUtils.c`. If you're curious about how the windows in this tutorial work, take a look at that file.

The first function, `SetUpWindows` just sets up the console and drawing windows.

The next part of the program in the `while` loop actually does the drawing. The function `ChooseShape` asks you for the kind of shape to draw. If you choose to quit, `ChooseShape` returns `NULL`, and the program ends. Otherwise `ChooseShape` creates a shape and stores it in the variable `aShape`.

The next four function calls ask you for the coordinates of the shape and store them in four variables.

The next two lines are the most interesting lines of the program. The line

```
aShape->SetBounds(top, left, bottom, right);
```

sends a SetBounds message to the shape aShape to give it its coordinates. The line

```
aShape->Draw();
```

sends aShape a message to draw the shape.

"Sending a message" is object-oriented programming terminology for calling a method. You can think of "sending a message" as calling a function through an object.

The important thing to notice is that the program uses the same two lines to set the coordinates and to draw the shapes regardless of the kind of shape. The statement that draws the shape doesn't need to know what kind of shape it will draw. It just sends the shape object a Draw message. The object knows how to draw itself.

How Do Objects Know How to Draw Themselves?

The best way to find out what's going on is to step through the program. If LearnOOP is asking you for a shape, type Q to make the program end.

To learn how to use the source-level debugger, see the tutorial Chapter 5, "Tutorial: Bullseye," and Chapter 12, "The Debugger," in the THINK C User Manual

Choose **Use Debugger** from the **Project** menu to turn on the source-level debugger. The "bug" column will appear along the left edge of the project window.

Note

Remember that you need to be running under MutiFinder or System 7.0 to use the THINK C source-level debugger. If you're running on a Macintosh with only one megabyte of RAM, and you can't use the debugger, just read along in this section.

Choose **Run** from the **Project** menu to start LearnOOP. Instead of launching LearnOOP, THINK C starts the source-level debugger. Click on the diamond

next to the line that sends the Draw message. Your debugger window should look like this:

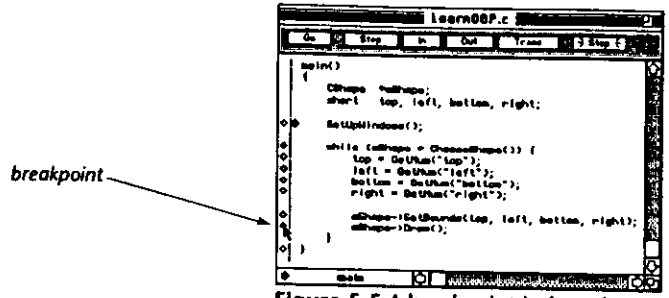

Figure 5-5 A breakpoint before the Draw message

Now click on the Go button in the debugger's source window.

THINK C starts the program again, and LearnOOP asks you to choose a shape. Type R for rectangle. LearnOOP asks you to type in the bounds for the shape. Type in whatever you like. If you can't think of anything, try 20 30 45 66. Be sure to press the Return key after each number.

As soon as you press Return after the last coordinate, LearnOOP stops at the line that sends a Draw message to aShape. To find out what's going on, click on the In button in the debugger's source window. You'll see the current statement arrow point to the beginning of a function called CRectangle::Draw():

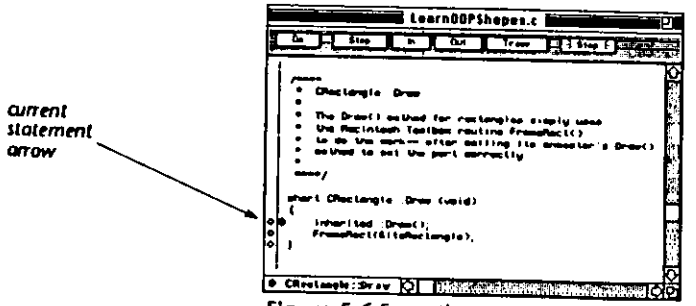

Figure 5-6 Execution stopped in CRectangle::Draw()

This procedure is actually a method. In fact, it's the method that handles Draw messages sent to objects of class CRectangle. In a moment, you'll see

what this class looks like. But first, see what happens when you choose a different kind of shape.

Click on the Go button in the debugger's source window to let LearnOOP draw the rectangle. When it asks you for another shape, type O for oval. Type whatever you want for the location. Because the breakpoint is still set, the program stops at the line that sends a Draw message to aShape. Click on the In button in the debugger's window again. This time you'll see the current statement arrow point to the beginning of another function:

current statement arrow

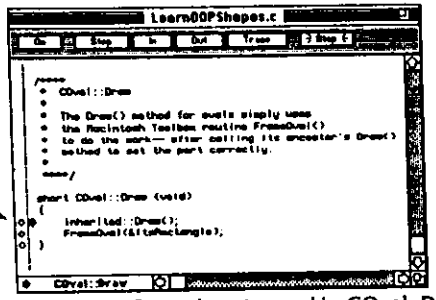

Figure 5-7 Execution stopped in COval::Draw()

The procedure `COval::Draw()` is the method that handles `Draw` messages for objects of class COval.

The next section explains how the shape-drawing classes are declared. You can keep exploring with the debugger. When you're ready to go on, type Q to quit the program.

The LearnOOP Class Declarations

The declaration of a class determines how an object responds to a message. The file `LearnOOPShapes.h` contains the class declarations for the shapes. To open `LearnOOPShapes.h`, choose the **Open...** command from the **File** menu, or hold down the Option key as you click in the title bar of the project window and choose `LearnOOPShapes.h` from the pop-up menu.

The class declarations in `LearnOOPShapes.h` look like this:

```
// CShape is the abstract class that
// describes all shapes. All shape
// classes are subclasses of this class.

struct CShape : indirect {
   Rect itsRectangle;    // The rectangle that
                         // encloses the shape.

   // The SetBounds method sets values
   // for itsRectangle. Most subclasses
   // won't need to override it.

   short SetBounds (short aTop, short aLeft,
                    short aBottom, short aRight);

   // The Draw method draws the shape. All
   // subclasses should override it.

   short Draw(void);
};

struct CRectangle : CShape {
   short Draw(void);
};

struct COval : CShape {
   short Draw(void);
};
```

These declarations in this file describe three classes. The first class, CShape, is a generic shape. Every object of class CShape has one instance variable: a rectangle called `itsRectangle`. Every object of class CShape also implements two methods: `SetBounds` to set the size of the shape, and `Draw` to actually draw the shape.

CShape is an **abstract class**. You use an abstract class as the common ancestor of a family of classes that share traits and behaviors. You never create

an object of an abstract class. Instead you create subclasses of the abstract class.

The common trait that all shapes share is a bounding rectangle. The common behaviors that all shapes share are being drawn and setting the bounding rectangle.

The next two classes, CRectangle and COval, are subclasses of CShape. They both inherit the `itsRectangle` instance variable and the `SetBounds` method. Both classes override the `Draw` method. This means that a CRectangle is identical to a generic shape, but it has its own drawing behavior. COval is also identical to a generic shape, but it has its own drawing behavior different from CRectangles.

A set of related classes is called a **class hierarchy**. Figure 5-8 shows you the class hierarchy for the LearnOOP program. The diagram shows that CRectangle and COval are descendants of the class CShape

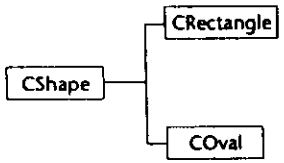

Figure 5-8 The LearnOOP class hierarchy

Now look at the implementation of the methods. The methods are defined in the file `LearnOOPShapes.c`. Use the **Open...** command in the **File** menu or double-click on the name `LearnOOPShapes.c` in the project window to open the file.

These are the method definitions for the abstract class CShape:

```
short CShape::SetBounds (short aTop,
                         short aLeft,
                         short aBottom,
                         short aRight)
{
   itsRectangle.top = aTop;
   itsRectangle.left = aLeft;
   itsRectangle.bottom = aBottom;
   itsRectangle.right = aRight;
}

short CShape::Draw (void)
{
   SetPort(DrawingWindow);
}
```

The first method, `SetBounds`, just sets up the boundary rectangle `itsRectangle`.

The second method is the generic `Draw` method. Because all shapes in this program appear in the drawing window, the `Draw` method sets the drawing port to be window stored in the global variable `DrawingWindow`. Every subclass of CShape uses CShapes `Draw` method as well as implementing its own.

Because CShape is an abstract class, there shouldn't be an object of class CShape. Every subclass of CShape should override the `Draw` method to draw a particular kind of shape.

The `Draw` methods for CRectangle and COval override the generic `Draw` method with the specific behavior for their class.

```
short CRectangle::Draw (void)
{
   inherited::Draw();
   FrameRect(&itsRectangle);
}


short COval::Draw (void)
{
   inherited::Draw();
   FrameOval(&itsRectangle);
}
```

These are the two functions you saw when you stepped through the program. They're what give CRectangle and COval their unique drawing behav-

ior. First they call `inherited::Draw()` to invoke the superclass's `Draw` method to set up the drawing window. Then they call the appropriate Macintosh Toolbox routine.

If you send an object of class CRectangle a `Draw` message, the `CRectangle::Draw()` method is called. First, the method invokes CShape's `Draw` method to set up the drawing window. Then it calls the Macintosh Toolbox routine `FrameRect` to draw the rectangle. If you send an object of class COval a `Draw` message, the `COval::Draw()` method is called.It invokes CShape's `Draw` method, the `FrameOval` procedure draws an oval.

How Do You Create an Object?

The specific `Draw` method that gets called when you send an object a `Draw` message depends on the class of the object. So, how do you create objects of different classes?

Look at the `ChooseShape` function in the file `LearnOOP.c`. This is the function that asks you to choose the shape:

```
CShape *ChooseShape (void)
{
   char  c;

   while (1) {
      c = GetCh("Create [O]val or [R]ectangle");

      switch (c) {
         case 'O':
         case 'o':
            return NULL;

         case 'R':
         case 'r':
            return new CRectangle;

         case 'O':
         case 'o':
            return new COval;

         default:    /* stay in the loop */
            break;
      }
   }
}
```

After it asks you for the kind of shape to draw, this function looks at what you typed and returns an object of the appropriate class.

If you typed an `R` or an `O`, it uses the operator `new` to create an object. When the "THINK C + Objects" option in the "Language Extensions" page of the Options... dialog is on, the word `new` becomes a keyword that creates an object of the named class.

Note that the return type of `ChooseShape` is CShape. The rules for object assignment say that you can assign an object variable of a subclass to an object variable of a superclass. What this means is that you can assign a CRectangle object to a variable of class CShape, but you can't assign an object of class CShape to variable of class CRectangle.

After it creates the shape, `ChooseShape` returns the new object, and the object gets set to the global variable `aShape` in the `while` loop in `main()`:

```
while (aShape = ChooseShape()) {
```

The type of `aShape` is CShape. Doesn't this mean that when you send a `Draw` message to `aShape` it should call the generic `CShape::Draw()` method? No, because the class of an object is determined at run time. In other words, the type of the variable aShape is CShape, but the class of the object that it points to is either CRectangle or COval. The class is determined while your program is running. This is called **dynamic binding**.

So if you create an object of class CRectangle and send it a `Draw` message, the method that gets called is `CRectangle::Draw()`.

What happens when you send the `SetBounds` message? Since neither the CRectangle class nor the COval class override the `SetBounds` method, the generic `CShape::SetBounds()` method gets called when you send either class of object a `SetBounds` message. This is called **inheritance**. The CRectangle and COval classes inherit the `SetBounds` method from their superclass, CShape.

How Do You Define a New Class?

Now, try creating a new class that's a subclass of one of the existing three classes. Here's how to create a subclass to drawsquares.

The first thing to do is to decide which class to subclass. Since a square is just like a rectangle— except that it has sides of equal lengths— try making the square class a subclass of CRectangle. You can use the same drawing method, but you'll have to override the `SetBounds` method. Here's how you'd declare the new CSquare class:

If you want to follow along, put this class declaration right under the COval declaration in the file `LearnOOPShapes.h`

```
struct   CSquare : CRectangle {
   short SetBounds (short aTop,
                    short aLeft,
                    short aBottom,
                    short aRight);
};
```

And here's one way to define the `SetBounds` method for the CSquare class:

If you want to follow along, put this class declaration right under the `COval::Draw()` *definition in the file* `LearnOOPShapes.c`

```
short CSquare::SetBounds (short aTop,
                          short aLeft,
                          short aBottom,
                          short aRight)
{
   aRight = aLeft + (aBottom - aTop);
   inherited::SetBounds(aTop, aLeft,
                        aBottom, aRight);
}
```

This function says to reset the right edge of the bounding rectangle to the same distance from the left edge as the bottom edge is from the top edge. (It ignores whatever value you give it for `aRight`.) The next line says to use the superclass' `SetBounds` method. In this case, it means to call CRectangle's `SetBounds` method which it inherits from CShape.

The only thing left to do is to edit the `ChooseShape` function in the file `LearnOOP.c` to handle the new shape. Here's how:

```
CShape *ChooseShape (void)
{
   char  c;

   while (1) {
      c = GetCh("Create Oval, Rectangle, Square");

      switch (c) {
         case 'Q':
         case 'q':
            return NULL;

         case 'R':
         case 'r':
            return new CRectangle;

         case 'O':
         case 'o':
            return new COval;

         case 'S':
         case 's':
            return new CSquare;

         default:    /* stay in the loop */
            break;
      }
   }
}
```


Try running the program now with the new class. When you ask `ChooseShape` to create a square, it should ignore the value you give for the right edge of the bounding rectangle.

Note

Because of the way LearnOOP asks for the coordinates of the bounding rectangle, you still have to provide a value for the right edge.

But How Does It Know?

You may still be wondering what in the LearnOOP program is calling the right method for a given message. There is a mechanism that figures out which method goes with which message for a given object. But this mechanism is abstracted away in the compiler and is part of the language.

Here's an example. You probably know that most microprocessors use different machine language instructions for multiplying integers and for multiplying floating point numbers. But when you write a C program you know that you can write a statements like:

```
floatVar = 1961.0102 * floatVal;
count = 24 * intVal;
```

Though there's a "hidden mechanism" that knows when to use floating point multiplication and when to do integer multiplication, you don't worry about it. The inner workings of the multiplication operator are abstracted away and are part of the language.

It's the same thing with object-oriented programming. The mechanism that matches messages and methods with objects is a part of the language. The power of object-oriented programming comes not from how it works but from the layer of abstraction that it adds to the low level mechanics of data structures and algorithms.

Where to Go Next

Since this small program declares only three classes, this is a good time to learn how to use the Class Browser. Chapter 6, "The Class Browser," describes the Class Browser in detail.

If you want, you can try adding more methods and classes. Try adding a method that asks where to put the shape so you don't have to provide an extra value for the CSquare class. Or you might want to try adding methods for filled shapes or for shapes with different line thicknesses.

The Class Browser 6

This chapter shows you how to use the class browser. The class browser is a tool that displays all the classes defined in your project as a tree chart. You can use the Class Browser to look at the declaration of classes or to find the definitions of methods.

Contents

Using the Class Browser 81
Finding the Declaration of a Class 81
Finding a class declaration with the Class Browser 81
Finding a class declaration with the editor 82
Finding the Definition of a Method 83
Finding a method definition with the Class Browser 83
Finding a method definition with the editor 84
Keyboard Shortcuts in the Class Browser 85
Class Browser Summary 86

Using the Class Browser

To use the browser, choose the **Browser** command from the **Source** menu. The Class Browser is shown in Figure 6-1.

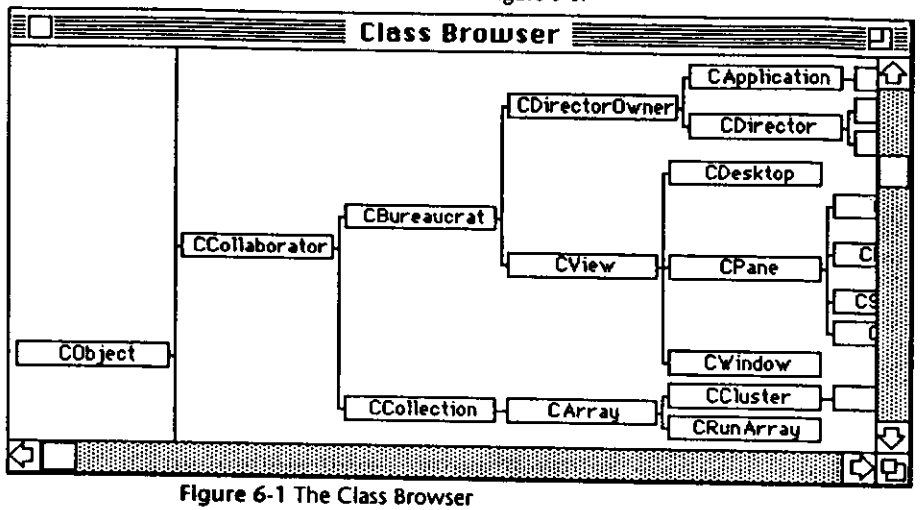

Figure 6-1 The Class Browser

The boxes represent the class hierarchy of the classes in your project. Each box represents one class. Subclasses are connected to their superclasses like an organizational chart.

In this picture, you can see that the classes CDesktop, CPane, and CWindow, are all descendants of CView.

Finding the Declaration of a Class

You can find a class declaration two ways: with the Class Browser or with the editor.

Finding a class declaration with the Class Browser

To find the declaration of a class, double-click on the box that represents the class. The editor opens the file that contains the class declaration and scrolls to the declaration of the class.

For instance, to see the declaration of the CBureaucrat class, double-click on the CBureaucrat box. The editor opens the file `CBureaucrat.h` and scrolls to the declaration of CBureaucrat, as shown in Figure 6-2.

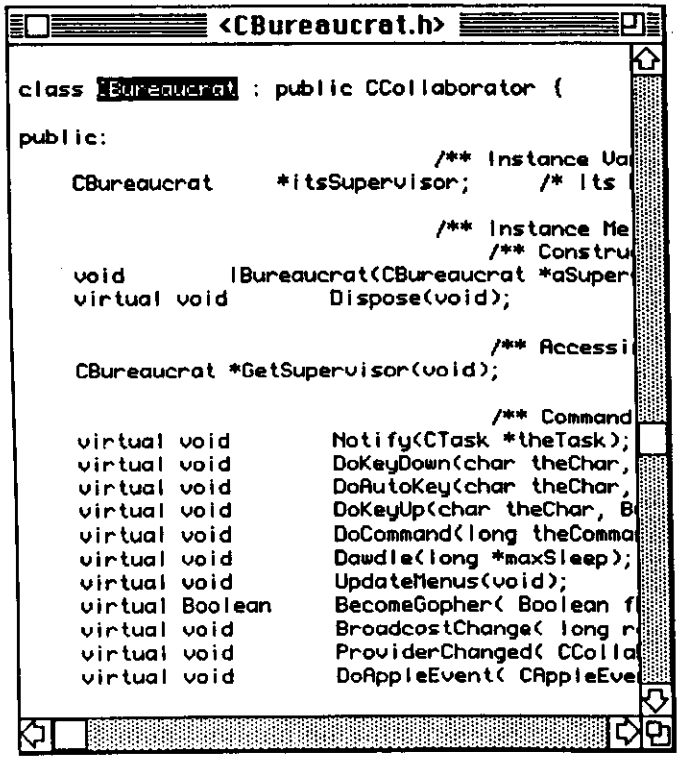

Figure 6-2 The declaration of CBureaucrat

Finding a class declaration with the editor

You can find the declaration of a class in the editor the same way you find the definition of function. When you hold down the Option key as you double-click on the name of a class, the editor opens the file that declares that class.

For example, if you're looking at the `CEditPane::IEditPane` method, and you want to see the declaration of the CBureaucrat class, you can Option-double-click on the name CBureaucrat in the parameter list.

Finding the Definition of a Method

You can find the definition of a method two ways: with the Class Browser or with the editor.

Finding a method definition with the Class Browser

To find all the methods that a class defines or overrides, hold the mouse down on the class name. A pop-up menu appears next to the class.

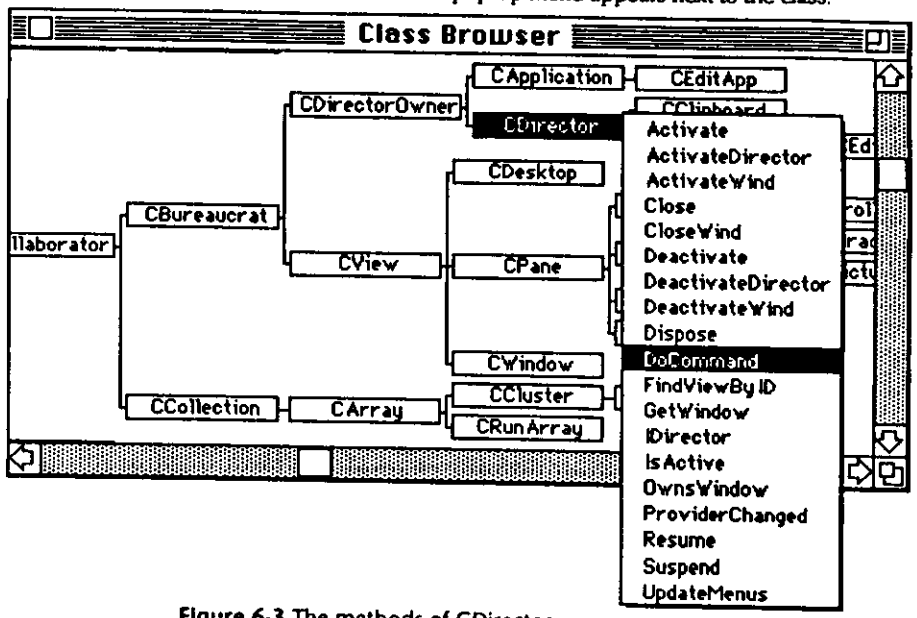

Figure 6-3 The methods of CDirector

If you choose a method from the pop-up menu, the editor opens the file that defines the method.

Note

The pop-up menu shows only the methods that a class defines or overrides. It does not include the methods that the class inherits from its superclass.

Finding a method definition with the editor

You can find the definition of a method from the editor the same way you can find the definition of a function. When you hold down the Option key as you double-click on the name of a method, the editor opens the file that defines that method and scrolls to the method definition.

Since several classes may define or override the same method, it's not clear which file the editor should open. When a method is defined in more than one class, the editor opens the Class Browser window and highlights the classes that define the method with a bold border.

For example, suppose that you Option-double-click on the method name `ChangeSize` in the `CEditPane::IEditPane` method, like in Figure 6-4.

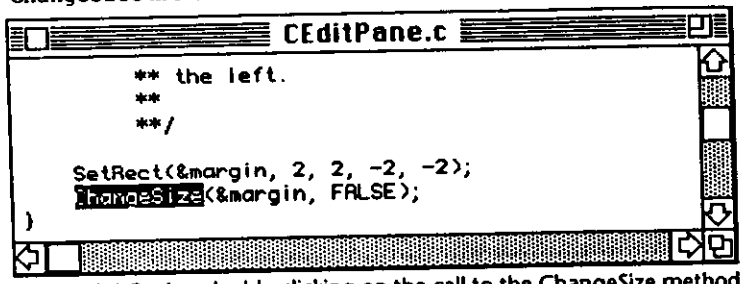

Figure 6-4 Option-double-clicking on the call to the ChangeSize method.

Since several classes define the `ChangeSize` method, THINK C opens the Class Browser window and highlights the classes that define the method as shown in Figure 6-5.

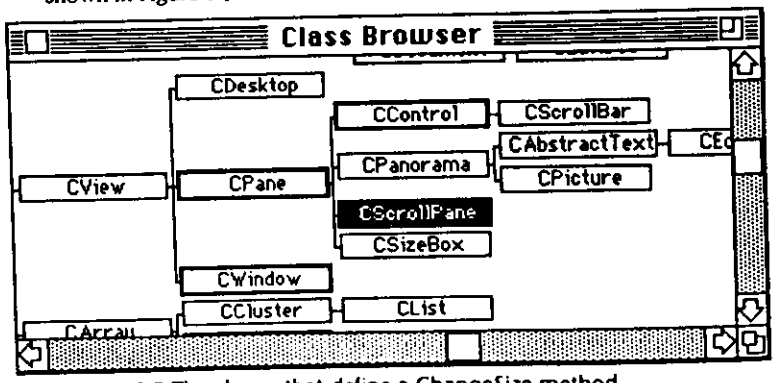

Figure 6-5 The classes that define a ChangeSize method.

To see the definition of the method, double-click on one of the highlighted classes. The editor opens the file that defines that method.

If you picked the wrong class or if you want to see another definition of the method, you can Option-double-click on the method name again to display the Class Browser window again.

Keyboard Shortcuts in the Class Browser

You can use the keyboard to navigate through the classes in the Class Browser window. The keyboard commands let you select classes or find the definitions of classes without using the mouse.

If you type the name of a class, the Class Browser highlights the class whose name matches what you've typed so far. For instance, as you type "CControl", the browser highlights CObject, CChore, CCollaborator, and finally CControl.

This is what other keys do in the Class Browser:

Key	Action
Enter or Return	Open the file that contains the definition of the selected class. This is the same as double clicking on a class name.
Up Arrow	Select the previous sibling of the current class.
Down Arrow	Select the next sibling of the current class.
Left Arrow	Select the superclass of the current class.
Right Arrow	Select a subclass of the current class.
Tab	Traverse each class in the class hierarchy.

Class Browser Summary

This table summarizes how to use the Class Browser:

If you're in the...	and you want to find a...	do this:
Class Browser	class declaration	double-click on the class name.
Editor	class declaration	Option-double-click on the class name.
Class Browser	method definition	hold down the mouse to see a pop-up menu of all the methods the class defines or overrides. Choose a method name from the pop-up menu to see its definition.
Editor	method definition	Option-double-click on the method name. If there is more than one definition, the Class Browser highlights all the classes that define the method. Double-click on the class name to see the definition of the method.

THINK C

The THINK Class Library

Part Three

7 The THINK Class Library
8 Exception Handling
9-70 The Classes
71 TCL Library Routines
72 Global Variables

The THINK Class Library

7

The THINK Class Library is a collection of classes that implement a standard Macintosh application. If takes care of things like handling menu commands, updating windows, dispatching events, dealing with MultiFinder, maintaining the Clipboard, and so on.

Contents

Introduction . . . 91
What you should know . . . 91

Conventions . . . 91
Types . . . 91
Naming conventions . . . 92
Object-oriented terminology . . . 92

Overview . . . 93
The class hierarchy . . . 93
The visual hierarchy . . . 95
The chain of command . . . 96
The flow of control . . . 97

Writing an Application with the TCL . . . 98
Creating the project in THINK Pascal . . . 99
Creating the project in THINK C . . . 100
Creating the application subclass . . . 100
Creating the document subclass . . . 101
Creating the pane subclass . . . 102

Working with Panes . . . 103
Windows and panes . . . 104
Coordinate systems . . . 105
Drawing in a pane . . . 106
Properties of panes . . . 107
Panoramas . . . 109
Scroll panes . . . 112
Cursor tracking . . . 113
Initializing views from resources . . . 113

Working with Menus . . . 114
Using MENU resources . . . 115
Building menus on the fly . . . 116

Dimming and checking menu items 116
Handling Low Memory Situations 118
Undoing and Mouse Tracking 119
Undoing 119
Mouse tracking 120
Segmentation and the THINK Class Library 120
Debugging and the THINK Class Library 121
Debugging aids in THINK C 121
Debugging aids in THINK Pascal 122
THINK Class Library Resources 122
Alerts 122
Controls 123
Error message strings 123
Menus 123
Menu bars 124
Small icon 124
Strings and string lists 124
Window template 125
Modifying the THINK Class Library 125
Where to Go Next 126

Introduction

This chapter introduces you to the THINK Class Library and talks about some of the general topics in application building. After you read this chapter, you should try running the demonstration applications. Be sure to follow the instructions in Chapter 2 of your user manual, and in this manual's Chapter 2, "Installing the THINK Class Library" to installs the THINK Class Library and its demonstration programs.

What you should know

You should be comfortable working with THINK C or THINK Pascal. You should be familiar with the fundamentals of Macintosh programming. Particularly, you should know about QuickDraw, the Window Manager, and the Memory Manager.

If you're using THINK C, you should be familiar with the THINK C implementation of objects. If you're using THINK Pascal, you should know Object Pascal. Earlier chapters in this manual cover these topics in detail. If you need a fundamental introduction to object-oriented programming, see Chapter 3, "Object-Oriented Programming." As you go through this chapter, remember that this is all new territory, so don't be discouraged if it takes a while for all the pieces to fall into place.

Conventions

This manual is designed for both the THINK C and THINK Pascal versions of the THINK Class Library. The chapters that follow describe the classes in the THINK Class Library. The chapters are organized like this:

Section	Description
Introduction	A brief description of the class.
Heritage	The class's superclass and its subclasses.
Using	A detailed description of how to use the class, along with examples.
Variables	A list of the instance variables of the class.
Methods	A description of each method that the class implements or overrides.

Types

Most of the types used to declare instance variables are the same in C and in Pascal. For 32-bit integers and 16-bit integers, this manual uses the Pascal names. C programmers should assume the corresponding C type:

Pascal type	C type
`integer`	`short`
`longint`	`long`

Hexadecimal values use C syntax: `0x0FA` is the same as `$0FA` in Pascal.

In Pascal a pointer that doesn't point to anything has a special value called NIL. In C the same value is called NULL. This manual uses both terms.

Naming conventions

The THINK Class Library follows these naming conventions:

Name	Description
CName	The name of a class.
OName	The name of a class that was defined in the original version of the THINK Class Library, but replaced in this version.
aName	A formal parameter.
cName	A class variable.
fName	A flag. Usually a Boolean instance variable.
gName	A global variable.
kName	A constant. In Pascal these are defined as const. In C these are defined with `#define` or enum.
itsName	An instance variable.
theName	A variable. Usually a local variable or an instance variable. Sometimes used for formal parameters.
macName	A Macintosh data structure used either as an instance variable or as a local variable.

Object-oriented terminology

This manual uses the terminology established in Chapter 3, "Object-Oriented Programming" to talk about object-oriented programming.

The syntax for calling a method inherited from the superclass is slightly different between C and Pascal, so this manual uses the phrase "call the inherited method." In THINK C, you would call it like this:

```
inherited::MethodName();
```

In THINK Pascal, you'd call it this way:

```
inherited MethodName;
```


Overview

The THINK Class Library is organized into three distinct, interacting structures: the class hierarchy, the visual hierarchy, and the chain of command. The class hierarchy is the set of all the classes that make up the THINK Class Library. The visual hierarchy describes the organization of all visible entities. The chain of command specifies which objects get to handle commands.

The THINK Class Library converts Macintosh events into **visual messages** and **direct commands**. A visual message is an event that affects the visual hierarchy. Mouse clicks, activate events, and update events are all visual messages. A direct command is a request that an object perform an action. Direct commands are usually the result of menu commands.

To convert Macintosh events into messages and commands, the THINK Class Library uses an object called a **switchboard**. The switchboard calls `GetNextEvent` or `WaitNextEvent` repeatedly and, depending on the kind of event, sends a message to the appropriate object. Each application has only one instance of a switchboard object. Another object, called the **bartender**, takes care of converting menu selections into direct commands.

The class hierarchy

The class hierarchy describes the relationships among all the classes in the THINK Class Library. All of the classes are descendants of the root class CObject. The class hierarchy is in Figure 7-1 on page 94. To make the drawing easier to read, the initial 'C' of each class is omitted.

Note

The classes with a heavy outline constitute the core classes that must be present in any program that uses the THINK Class Library.

Do not think of the class hierarchy as a functional description of the THINK Class Library. You might think from the drawing of the class hierarchy that the CRadioGroup class somehow interacts with the CBureaucrat class, but that's not the case.

Instead, think of the class hierarchy as a family tree. Each class inherits all of the behavior (methods) and all of the attributes (instance variables) of its ancestor. So the CButton class *is* a control. It behaves like a control and responds to all of the messages a control responds to.

Some of the classes in the class hierarchy are **abstract classes**. The THINK Class Library never creates objects of these classes. They're used to give a

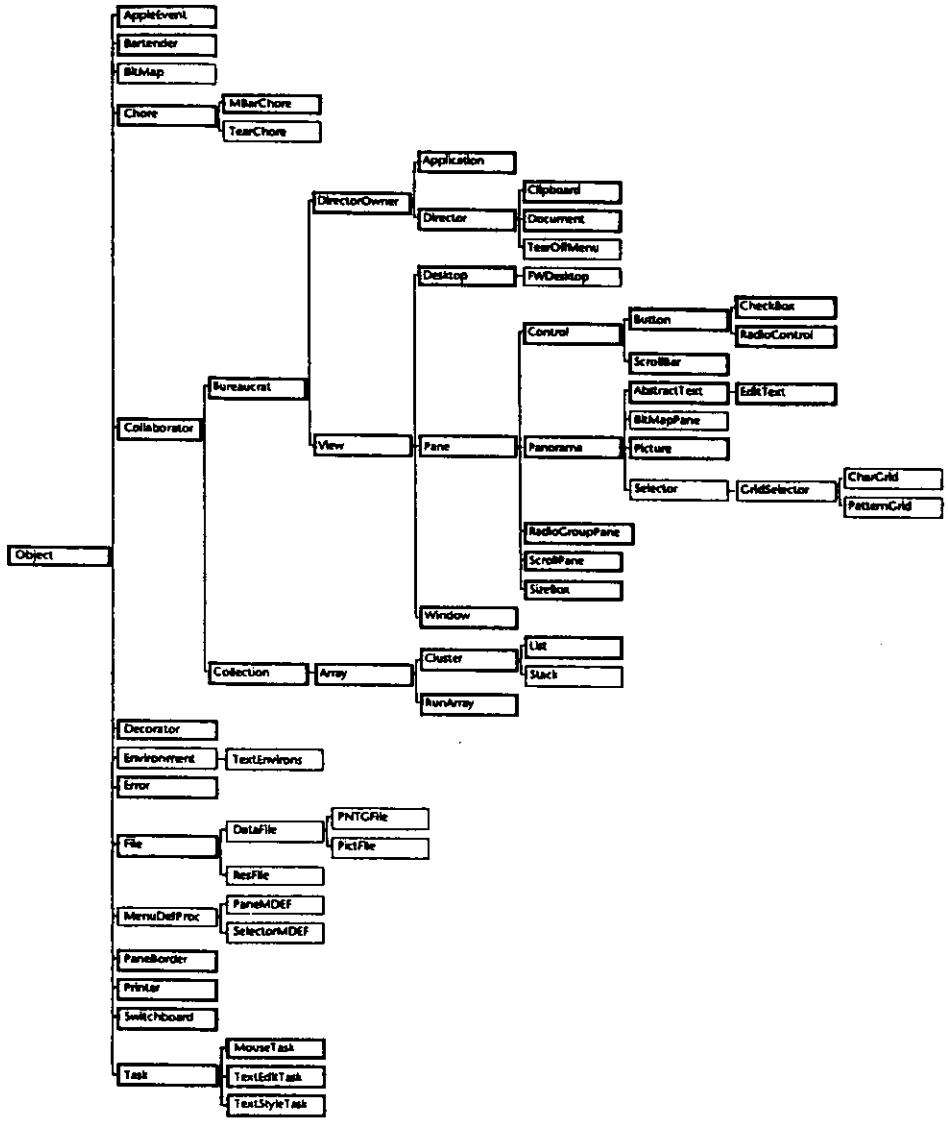

Figure 7-1 The THINK Class Library's class hierarchy

family of objects common behaviors. The most important abstract classes in the THINK Class Library are CObject, CCollaborator, CBureaucrat, CView, CDirectorOwner, and CApplication.

The visual hierarchy

The visual hierarchy describes all the visible objects, or **views**, that the THINK Class Library knows about. The visual hierarchy is built around the idea of **enclosures**. Everything that you see on the screen belongs to—is enclosed by—another visual entity.

At the top of the visual hierarchy is the **desktop**. The desktop encloses all of the windows in an application. Each window encloses one or more panes, and those panes may enclose other panes. Figure 7-2 shows a typical visual hierarchy.

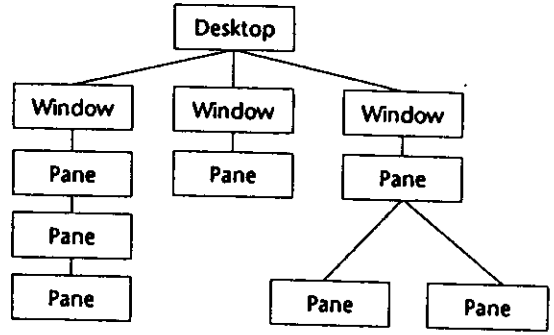

Figure 7-2 A typical visual hierarchy

Unlike the class hierarchy, the visual hierarchy is dynamic. It changes as your program runs. When you open a new document, you add another window to the desktop, and you add another set of panes to the new window.

Panes are the most important kinds of views. All drawing takes place in a pane. The THINK Class Library defines different kinds of panes designed for different kinds of displays. Every pane has its own drawing environment, so you can draw in a pane without worrying about where it is on the screen.

Visual events work their way down the visual hierarchy. Since Macintosh update and activate events always have a window associated with them, these messages get sent directly to a window object. Mouse clicks and cursor adjustment messages always work their way down from the desktop to the active window to the appropriate pane.

Panes can handle visual commands like mouse clicks. When you click in a pane, the switchboard determines which pane the mouse went down in and sends it a `DoClick` message. Because they're descendants of CBureaucrat, panes can be in the chain of command and respond to direct commands.

The chain of command

The chain of command specifies which object handles a direct command. The chain of command is based on the idea of **supervisors** and **bureaucrats**. Every object in the chain of command is a bureaucrat. If a bureaucrat can't handle a direct command, it passes the command on to its supervisor. The application is the only bureaucrat that does not have a supervisor. If the application doesn't handle the command, no object will.

Objects in the chain of command are descendants of the class CBureaucrat. Every bureaucrat has a `DoCommand` method that the subclass can override to handle specific commands. The default `DoCommand` method just sends a `DoCommand` to its supervisor.

The first object to get a chance to handle a command is called the **gopher**. If the object that the gopher points to can't handle the command, it sends the command on to its supervisor. Your application is responsible for sending a `BecomeGopher` message to a bureaucrat that should be the gopher.

In Figure 7-3, the gopher points to a pane whose supervisor is a document. If the pane can't handle a direct command, it passes the command on to the document. If the document can't handle the command, it gets passed up to the application.

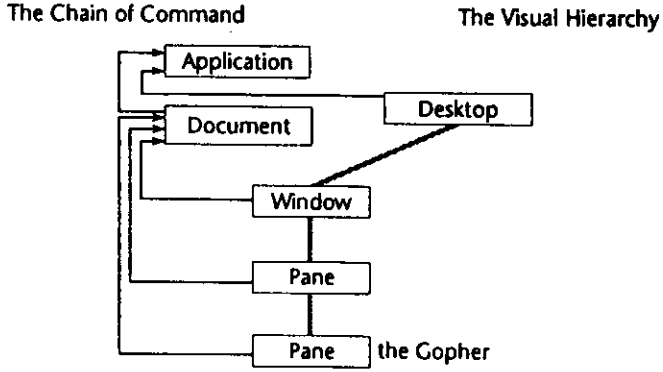

Figure 7-3 The gopher, the chain of command, and the visual hierachy

The enclosures in the visual hierarchy are drawn in gray. Note that the chain of command is separate from the visual hierarchy.

CDirector is an important subclass of CBureaucrat that you need to know about. A director is a bureaucrat that supervises a window. Directors handle the communication between the visual hierarchy and the chain of command. For instance, when a window gets an activate event, it sends an `ActivateWind` message to its supervisor, which is always a director. The director can then take some action as a result of becoming active.

Another descendant of CBureaucrat is CDocument. A document is a director that has a file associated with it. Documents manage the communication between windows, files, and menu commands. The default document class handles common commands like **Save**, **Save As...**, **Print**, etc. You can think of a document as a file that you view through a window. A better way to think about a document is that it is the essence of a Macintosh application. It is anything that you can display and manipulate inside a window.

Every bureaucrat is a descendant of the class CCollaborator. A **collaborator** is an object that can let other objects know that something has happened. One collaborator is called the **provider**, and another is called the **dependent**. For instance, a provider might be a document that displays data in several windows. Its dependents might be the windows. If the data changes, the provider lets the dependents know, so the windows redisplay the data correctly.

The flow of control

The chain of command and the visual hierarchy get their direction from the switchboard. The switchboard gets events from the Macintosh Event Manager and converts the event into messages for either the chain of command or the visual hierarchy. Messages for the chain of command usually go to the gopher, the first bureaucrat in the chain. Messages for the visual hierarchy go to the active window or to the desktop.

When you press the mouse, the switchboard sends a `DispatchClick` message to the desktop. If the click was in the menu bar, the desktop sends the bartender a message to update the state of the menus before they appear. The bartender sends a message to the chain of command to enable and disable the appropriate menu items. Then the desktop uses the bartender to convert the menu selection into a direct command and sends a `DoCommand` message to the gopher.

If the click was in a window, the desktop sends a `DispatchClick` message to the window, which eventually sends a `DoClick` message to the

pane the mouse went down in. The pane's `DoClick` method can then do whatever's appropriate for the pane. It might even send a `DoCommand` message to an object in the chain of command.

When the switchboard gets an activate or an update event, it sends an `Activate`, `Deactivate`, or `Update` message to the window. The window sends a similar message to its director.

When you type, the switchboard sends a `DoKeyDown` or a `DoAutoKey` message to the gopher. If you hold down the Command key when you type, the switchboard asks the bartender to convert the key into a command and sends a `DoCommand` message to the gopher.

If you're running under MultiFinder, and you bring another application to the foreground, the switchboard sends a `Suspend` message to the application (not the gopher) which sends `Suspend` messages to all of its directors. A similar thing happens when your application comes to the foreground.

Note

The THINK Class Library treats desk accessories as if they were in their own layer, even if you're not using MultiFinder, so your application will still get suspend and resume "events" when you bring up a desk accessory.

Writing an Application with the TCL

This section tells you how to write an application with the THINK Class Library. The easiest way to do this, is to take the Starter program and build from it.

To create an application with the THINK Class Library, you create subclasses of existing classes. The three classes you need to override are CApplication, CDocument, and CPane. Your application subclass determines the overall structure of your application. The document subclass implements the way your application handles its files, and the pane subclass implements how the information in your file appears in the document windows.

In addition to the subclasses, you also need a resource file for your project. This resource file must contain the standard THINK Class Library resources as well as your own. The standard THINK Class Library resources are in the file `TCL Resources`. When you use the THINK Class Library, make a copy of this resource file and name it the same as your project and append `.rsrc` to it. Then add your own resources to it.

Note

For more information about resource files and the THINK Class Library, see "THINK Class Library Resources" on page 122. To learn about using resources with a project, see Chapter 7, "The Project," of the *THINK C User Manual* or Chapter 7 "Working with Projects" in the *THINK Pascal User Manual*.

Creating the project in THINK Pascal

Use the Starter project as the beginning of your project. The Starter project is in the `Starter Folder` in the `TCL 1.1 Demos Folder`.

1. Copy the `Starter Folder` and change its name to the name of your application
2. Open the new folder
3. Rename `Starter.π` to the name of your application
4. Rename Starter.Build.π to the nameof your application, but keep the .Build.π suffix.
5. Rename `Starter.π.rsrc` to the name of your project plus `.rsrc`
6. Open the project without the .Build.π suffix.
7. Use the **Run Options...** dialog to choose the resource file you renamed in step 5.
8. Use the **Find...** command in the **Search** menu to change `Starter` to the name of your application
9. Use the **Save As...** command to save the files under your own names (the **Save As...** command also changes the names of the files in the project)

THINK Pascal uses two projects—one for debugging and one for building the final application. When you build an application, THINK Pascal has to turn the debugging options off. That means that the entire THINK Class Library would need to be recompiled. By using two projects and switching to the `.Build.π` project when you want to build the final application, only your own files will need to be recompiled.

Creating the project in THINK C

Use the Starter project as the beginning of your project. The Starter project is in the `Starter Folder` in the `TCL 1.1 Demos Folder`.

1. Copy the `Starter Folder` and change its name to the name of your application
2. Open the new folder
3. Rename `Starter.π` to the name of your application
4. Rename `Starter.π.rsrc` to the name of your project plus `.rsrc`
5. Open the project
6. Use the **Find...** command in the **Search** menu to change `Starter` to the name of your application
7. Use the **Save As...** command to save the files under your own names (the **Save As...** command also changes the names of the files in the project)

Use the **Set Project Type...** dialog to make sure that your project is Multi-Finder Aware and will receive Suspend & Resume events.

Creating the application subclass

If your application is a standard Macintosh application, your application subclass must override these methods:

Initialization method	
`SetUpFileParameters`	`CreateDocument`
`OpenDocument`	`DoCommand`

Your initialization method should initialize any instance variables that your subclass declares. It must call CApplication's `IApplication` method.

The `SetUpFileParameters` method sets up the standard file parameters that specify which files are visible in the standard file box when the user chooses **Open...** from the **File** menu.

The `CreateDocument` method creates a new, untitled document. The document it creates is one that you define as a subclass of `CDocument`. After creating the document, your `CreateDocument` method should send it a `NewFile` message. This is the method that gets called when the user chooses **New** from the File menu.

The `OpenDocument` method is like the `CreateDocument` method. Instead of sending the newly created document a `NewFile` message, though, this document should send it an `OpenFile` message. The `OpenDocument` method takes one parameter, an `SFReply` record, that contains the information about the file that the user chose to open.

The `DoCommand` method handles all the application-specific commands. Most of the commands should be handled at the document level. Some commands, like **New**, **Open**, and **Quit**, are handled by the default application class.

Creating the document subclass

The document is where your application draws and displays its data. All documents have windows associated with them. Most documents also have an associated file. Neither the window nor the file are created automatically. You must create them yourself.

Your document class should override these methods:

Initialization method	`OpenFile`
`Dispose`	`DoSave`
`DoCommand`	`DoSaveAs`
`NewFile`	`Revert`

Your document class must have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be `I`*YourDoc* where *YourDoc* is the name of your document class. Your initialization method should call the inherited method. The supervisor of a document is always `gApplication`.

If your document allocates memory, you should also override the `Dispose` method to deallocate it. Be sure that your method calls `inherited::Dispose` (`inherited Free` in Pascal) to make sure that the document is disposed of properly.

Note

You do not need to dispose of the `itsWindow` or the `itsFile` instance variables. The default `Dispose` or `Free` method does that for you.

Your document class's `DoCommand` method does most of the work in your application. When a window is active, the switchboard will send all commands to the document first (it's the gopher), and if the document can't handle it, the application class tries to handle it. Your document class should handle all the commands it knows about, and call the inherited method when it can't.

Note

Be sure that your DoCommand method or that one of the methods it invokes sets the instance variable dirty to TRUE when there has been a change to the document.

Your document class will get a NewFile message when the user chooses **New** from the **File** menu. This method needs to create a window and attach the panes for it. The NewFile method doesn't need to create a file until the user tries to save the document.

Your document gets an OpenFile message when the user chooses **Open...** from the **File** menu. The OpenFile method has one argument: a pointer to a Macintosh SFReply record. When you get the OpenFile message, you can be sure that the SFReply record is properly filled in. Your OpenFile message needs to create an instance of a file object (usually of class CDataFile). You can send your file any of several read messages to obtain its contents. Your OpenFile method also needs to create a window to display the contents of the file, just as your NewFile method does.

When the user chooses **Save** from the **File** menu, your document gets a DoSave message. Your DoSave method should write the contents of its file to disk. The file object is stored in the instance variable itsFile.

When the user chooses **Save As...** from the **File** menu, your document gets a DoSaveAs message. This method takes an SFReply record, and you can be sure that it is properly filled in. Your document class needs to override this method to write its data to a file.

If your application supports the **Revert** command, you should implement it in the DoRevert method. Your implementation might do the same thing as closing without saving, then opening the file again.

Creating the pane subclass

Once you've created your windows and opened your files, you need to display them somewhere. In the THINK Class Library, you don't write directly onto the window. Instead, you create a subclass of class pane.

When you create a subclass of Pane, you need to override these methods:

I*YourPane*	Dispose
Draw	DoClick

Your pane class should have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be I*YourPane* where *YourPane* is the name of your pane class. Your initialization method should call the inherited method of whatever pane class you're overriding. The supervisor of a pane should be either the pane that encloses it or the director its window belongs to.

The pane initialization method is where you set the pane's location in its enclosure and its characteristics. If you want your pane to receive clicks, be sure to send the pane a SetWantsClicks(TRUE) message, otherwise mouse clicks in your pane are ignored.

If your pane allocates memory, you should also override the Dispose or Free method to deallocate it. Be sure that your method calls inherited method to make sure that the pane is disposed of properly.

The Draw message tells your pane to draw its contents. You can assume that the port, clip region, and coordinate system have been set up correctly. See section "Drawing in a pane" on page 106 to learn all about drawing in a pane.

When you click in a pane, it will get a DoClick message. Your DoClick method can handle the mouse click itself to draw something, to drag an object, or to select something. Some panes, like the edit text pane implemented by CEditText, have built-in DoClick methods.

If you want your mouse action to be undoable, you need to create a subclass of CMouseTask and send it in a TrackMouse message. The section "Mouse tracking" on page 120 goes into more detail about undoable mouse actions.

Working with Panes

In the THINK Class Library, all your drawing takes place in a **pane**. A pane is a rectangular region of the screen completely enclosed by a window. A window may have several panes, and each pane may have several subpanes. Each pane has its own drawing environment and can handle its own visual commands.

Every pane has an **enclosure** that completely encloses the pane. A pane also has a supervisor—an object in the chain of command. A pane's enclosure and supervisor can be the same object, particularly if the pane belongs to another pane. In most the common case, though, the supervisor is the director that owns the window or the pane.

Every pane has its own drawing environment. The rectangle that describes the edges of a pane is the **frame**. The frame defines a local coordinate sys-

tem for the pane. In most cases, the top, left corner of the pane is the point (0, 0).

Windows and panes

All panes are subclasses of the abstract class CView which defines the behavior of visual entities. A window is a view, but it is not a pane. Other visual entities are subclasses of panes; they include things like controls, borders, pictures, and size boxes.

Every pane belongs to a window or to another pane. The outer, or owning pane, is the enclosure. Every pane has one enclosure. For example, in the window in Figure 7-4, there are seven panes (the window is not a pane):

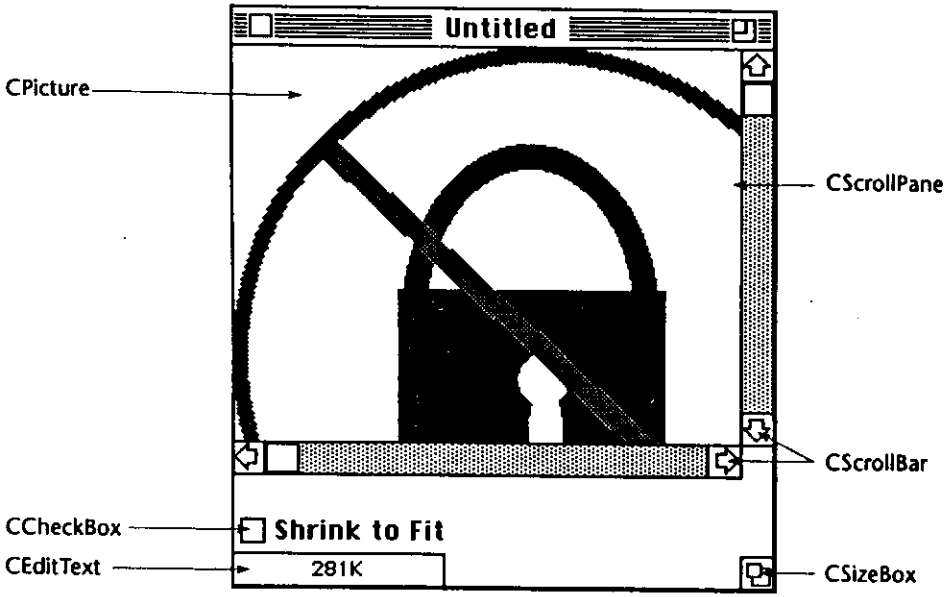

Figure 7-4 The panes in a window

The size box, the check box, the text, and the scroll pane are enclosed by the window. The two scroll bars belong to the scroll pane. The picture is enclosed by the window.

Coordinate systems

When you're working with panes in the THINK Class Library, you need to know about four coordinate systems:

- Global coordinates
- Window coordinates
- Frame coordinates
- QuickDraw coordinates

The desktop, some internal methods, and some Toolbox routines use **global coordinates**. In this coordinate system, all units are in pixels, and (0, 0) is at the top, left corner of the main screen. You rarely use global coordinate in the THINK Class Library.

In **window coordinates** the top, left corner of the window's content region is (0, 0) and each unit is a pixel. The only time you need to use window coordinates is to set the position of a pane whose enclosure is a window. Window coordinates are also useful as a common point of reference for two different panes in the same window.

Frame coordinates provide a local coordinate system for a pane. Units in frame coordinates are in pixels, and the point (0, 0) is usually the top left corner of the pane. If the pane moves within its enclosure, the coordinate system does not change; the top left corner is still (0, 0). The only time this origin point changes is when you scroll the pane. Each pane can choose to use long coordinates or short coordinates.

All drawing and mouse tracking is done in **QuickDraw coordinates**. This is the coordinate system that the Macintosh Toolbox uses for its drawing operations. QuickDraw coordinates are only valid after a call to `Prepare`.The relationship between QuickDraw coordinates and the other coordinate systems depends on whether a pane is using long frame coordinates or short frame coordinates.

Short coordinates map directly to QuickDraw coordinates. Each element in a short coordinate uses 16-bit values, so a pane that uses short coordinates is limited to the rectangle (–32768, –32768, 32767, 32767). **Long coordinates** layer a 32-bit coordinate system on top of the QuickDraw 16-bit coordinates. The long coordinate system lets you use a much larger coordinate area for your pane. Since all drawing takes place in QuickDraw coordinates you have to map the long coordinates to QuickDraw coordinates when you draw in a pane.

The CPane class defines several methods that transform coordinates from one system to another.

Drawing in a pane

Every pane has its own drawing environment. The rectangle that describes the edges of a pane is the pane's **frame**. The frame defines a local coordinate system for the pane. In most cases, the top, left corner of the pane is the point (0, 0).

To draw in a pane, you override its `Draw` method. The THINK Class Library sends your pane a `Draw` message whenever the pane needs to be updated. You can send a `Refresh` message yourself if you want to force an update event.

To draw in a pane use the standard QuickDraw routines. The pane's `Prepare` method sets up the QuickDraw port. If your pane uses short coordinates, you the coordinate system is set up correctly. If your pane uses long coordinates, you need to transform frame coordinates to QuickDraw coordinates before you draw. You can CPane methods `FrameToQD` and `FrameToQDR` convert frame points and rectangles to QuickDraw points and rectangles.

The THINK Class Library uses the types `LongRect` and `LongPt` for both long and short corrdinates. You'll notice that most of the descendants of CView that work with points and rectangles use these types.

Note

If you've worked with earlier versions of the THINK Class Library, this is the biggest change you'll notice since it will require some changes to your programs.

These two types are defined like this in `LongCoordinates.h` in C:

```
typedef struct LongPt
{
   long v, h;
} LongPt;

typedef struct LongRect
{
   long top, left, bottom, right;

} LongRect;
```


And like this in `TCL.p` in Pascal:

```
type
   LongPt = record
               case integer of
            1: (
               v, h: longint
               );
            2: (
               vh: array[VHSelect] of longint
               );
            end;

   LongRect = record
               case integer of
            1: (
               top, left,
               bottom, right: longint
               );
            2: (
               topLeft: LongPt;
               botRight: LongPt
               );
            end;
```

If the pane uses short coordinates, frame coordinates and QuickDraw coordiantes are identical, so the values stored in a `LongRect` or in a `LongPt` are in QuickDraw coordinates. To use them with QuickDraw routines, however, you'll need to convert them to the QuickDraw types Rect and Point. The THINK Class Library provides several utility routines to do these conversions. For a complete list, see "Long Coordinate Utilities" on page 488.

To draw directly in a pane as a result of a mouse click, you need to override the `DoClick` method of the pane. Just as in drawing, if your pane uses short coordinates, you do not need to transform frame coordinates to QuickDraw coordinates. If you use long coordinates, you must transform the coordinates before you draw. To make your mouse action undoable, use a mouse task. See the section "Undoing and Mouse Tracking" on page 119.

Properties of panes

When a pane moves or changes size, all of the panes that it encloses change as well. The way a pane changes depends on its **sizing characteristics**. When you create a pane, you specify its horizontal and vertical sizing characteristics.

The horizontal sizing characteristic specifies how the pane's left and right edges change.

Horizontal sizing	Meaning
`sizFIXEDLEFT`	The left edge of the pane is always the same number of pixels from the left edge of the enclosing pane as when it was originally placed.
`sizFIXEDRIGHT`	The right edge of the pane is always the same number of pixels from the right edge of the enclosing pane as when it was originally placed.
`sizFIXEDSTICKY`	The left and right edges stick to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane will scroll with it.
`sizELASTIC`	The width of the pane grows or shrinks by the same amount as the width of the enclosing pane.

The vertical sizing characteristic specifies how the pane's top and bottom edges change.

Vertical sizing	Meaning
`sizFIXEDTOP`	The top edge of the pane is always the same number of pixels from the top edge of the enclosing pane as when it was originally placed.
`sizFIXEDBOTTOM`	The bottom edge of the pane is always the same number of pixels from the bottom edge of the enclosing pane as when it was originally placed.
`sizFIXEDSTICKY`	The top and bottom edges stick to their original locations in the enclosing pane. If the enclosure scrolls vertically, the pane will scroll with it.
`sizELASTIC`	The height of the pane grows or shrinks by the same amount as the height of the enclosing pane.

A couple of examples might help: A vertical scroll bar in a window has the characteristics `sizFIXEDRIGHT` and `sizELASTIC`. It has a fixed horizontal length and remains anchored to the right edge of the window. Vertically, it changes with the height of the window. A status box in the lower left corner of a window would be `sizFIXEDLEFT` horizontally and `sizFIXEDBOTTOM` vertically. It has a constant size and remains anchored to the bottom left corner of the window.

In Figure 7-5, the dark line represents the window. It contains a main pane that takes up most of the window and several other panes. The two panes that hold the scroll bars are fixed to the edges of the window. When the window is resized, they grow or shrink by the same amount as the window. The panes that hold the grow box and the status box are anchored to the bottom corners of the window. The square pane in the upper left portion of the main pane will always be there, regardless of how the window changes. Its dimensions will not change automatically.

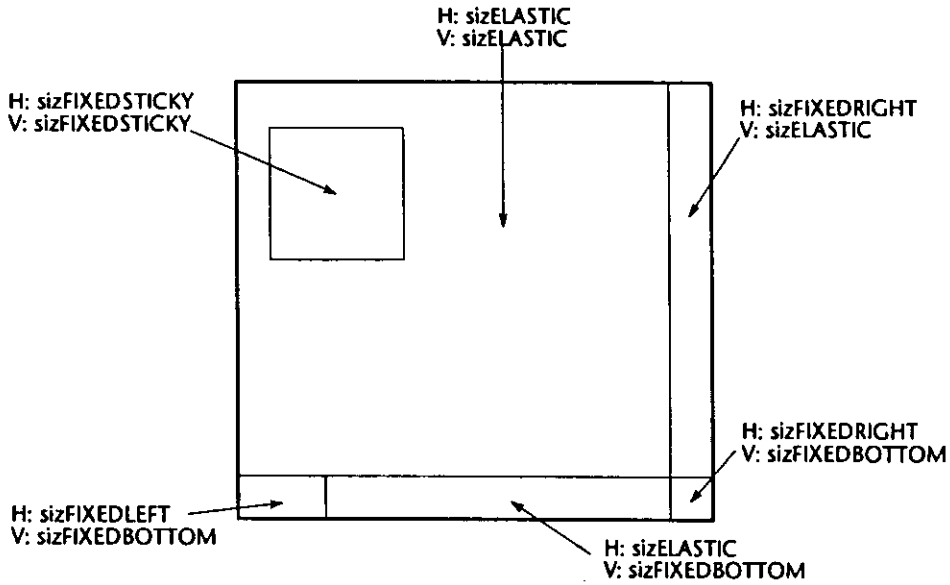

Figure 7-5 Horizontal and vertical sizing

Panoramas

Almost everything you want to display is bigger than a pane. Graphics and text, for instance, frequently take up more room than what you can fit in a pane. The THINK Class Library provides a **panorama** class, CPanorama, that lets you display portions of a large graphic in a pane. You might say a panorama is a pane that scrolls.

Think of a panorama as a sheet of paper glued to a desk. The frame moves over the paper. The only part of the panorama you can see is what's inside the frame. To scroll, you move the frame around on the panorama to see different parts of it.

The rectangle that completely encloses the panorama is called the **bounds rectangle**. The bounds rectangle defines the size and coordinate system of the panorama. Usually, the top, left corner of the bounds rectangle is the point (0, 0), and the units in its coordinate system are pixels.

The coordinate system of the bounds rectangle specifies how the frame moves over the panorama when you scroll. In the usual case, when you scroll up, you move the pane up a pixel. In some applications, though, you want to scroll more than one pixel. In a text editing application or in a spreadsheet application, you want to move up by an entire line.

You can specify a **scale** that says how many pixels make up a single panorama unit. You can set different scales for horizontal and for vertical units. In a graphics application, each unit might be one pixel. In a spreadsheet, a vertical unit might be 12 pixels, and a horizontal unit might be 60 pixels.

The units of the panorama bounds are for scrolling only. For drawing, you'll use the frame coordinates which are always pixel units.

There are two ways to talk about the top, left corner of the frame of the pane. The top, left corner of the pane, expressed in panorama units, is the **position** of the frame in the panorama. The top, left corner, expressed in frame coordinates, is the **origin** of the frame.

The key thing to remember is that scrolling always happens in panorama units. Drawing always takes place in frame coordinates. As you scroll, the origin of the frame changes. A couple of examples might help.

In Figure 7-6, both the horizontal and vertical scales are set to 1 pixel per panorama unit. The bounds rectangle of the panorama is (0, 0, 400, 380).

The portion of the picture you can see in the frame of the pane, the fish's tail, starts at (165, 210) in the panorama.

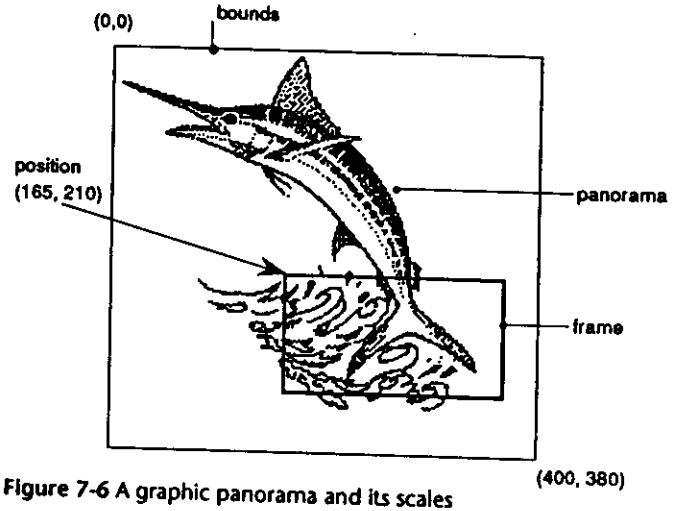

Figure 7-6 A graphic panorama and its scales

Since the panorama units match the frame units, the position of the frame in the panorama and the origin of the frame are the same. If you were to draw a line from (220, 230) to (270,230), it would cut across the tail of the fish.

In Figure 7-7, which shows a panorama with text, the horizontal scale is 6 pixels per unit, and the vertical scale is 12 pixels per unit. The bounds rectangle of the panorama is (0, 0, 40, 9). In the panorama scale, this means 8 lines of 40 characters each. The position of the frame in the panorama is (0, 3), the beginning of the fourth line. The origin of the frame, however, is at

(0, 36). If you wanted to draw a line to strike out the word "And," you would draw it from (0, 42) to (18, 42)

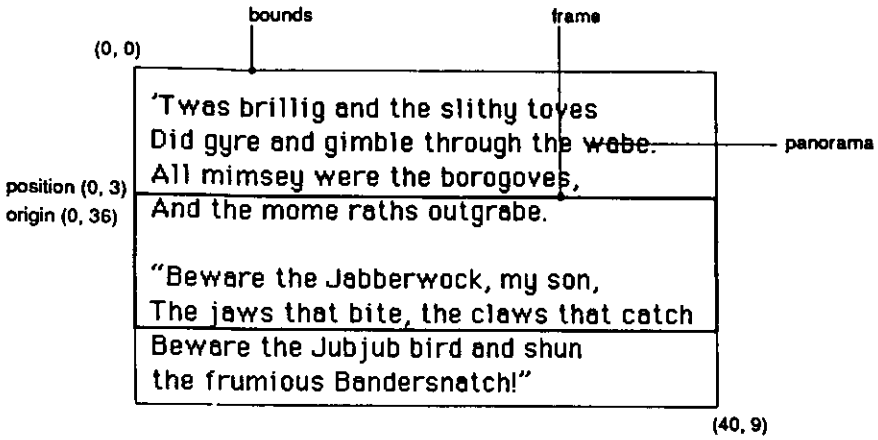

Figure 7-7 A text panorama and its scales

Note

The top, left of the bounds rectangle doesn't have to be (0, 0). You can define the bounds coordinate system that's most convenient for the kind of data you're displaying in the panorama.

Scroll panes

To make it easy to use panoramas, the THINK Class Library provides a class called CScrollPane that implements a **scroll pane**. Scroll panes give you an easy way to attach scroll bars to your panorama.

You create a scroll pane the same way as any other pane. You can request a vertical scroll bar, an horizontal scroll bar, and a size box. Then you use the `InstallPanorama` method to associate a panorama with the scroll pane. The scroll pane examines the panorama and adjusts the scroll bars appropriately. You can specify how many panorama units to scroll when you click on different parts of the scroll bar.

The scroll bars and the panorama communicate through the scroll pane. When you click in one of the scroll bars, it tells the scroll pane which tells the panorama how much to scroll.

Cursor tracking

The `AdjustCursor` method that all panes inherit from CView lets you change the cursor when it moves into your pane. Most of the time you'll use only one cursor in the whole pane. In this case, all you have to do is set the cursor with the Toolbox routine `SetCursor`. Look at the `AdjustCursor` method in CEditText for an example.

Sometimes, though, you might want to use different cursors within the same pane. The `AdjustCursor` method lets you do this as well, but it takes a little more work. See the description of the CView class in Chapter 52.

Initializing views from resources

The `IViewRes` method lets you initialize any descendant of CView from a resource template.

Resource	Class
`Pane`	CPane
`Pano`	CPanorama
`PctP`	CPicture
`ScPn`	CScrollPane
`AbTx`	CAbstractText, CEditText
`View`	CView

You can use ResEdit to create the resource templates for each class.

Note

Your THINK C package includes a file `TCL TMPLs` that contains `TMPL` resources you can install into ResEdit. These `TMPLs` let you create and edit the resources above. See the instructions in Chapter 2, "Installing the THINK Class Library," to learn how to install these `TMPLs` into ResEdit.

When you use ResEdit to create view resource templates, keep in mind these values for sizing and clipping mnemonics.

Sizing values	Clipping values
`sizFIXEDLEFT = 0`	`clipAPERTURE = 0`
`sizFIXEDRIGHT = 1`	`clipFRAME = 1`
`sizFIXEDTOP = 2`	`clipPAGE = 2`
`sizFIXEDBOTTOM = 3`	
`sizFIXEDSTICKY = 4`	
`sizELASTIC = 5`	

Working with Menus

The THINK Class Library lets you think of your menu commands more abstractly than the Macintosh Toolbox. Instead of identifying a menu command by its menu ID and item number, the THINK Class Library lets you assign unique **command numbers** to each item of the menu. With command numbers, you can reassign functions to different menu items without having to rebuild the application.

Command numbers are positive long integers in the range 1 to 2,147,483,647. Command numbers in the range 1 to 1023 are reserved for the THINK Class Library. Command number 0 is reserved for `cmdNull`, the null command. All other command numbers (1024 to 2,147,483,647) are available for your application.

The reserved commands are for the most common Macintosh application commands like **Open**, **Save**, **Quit**, **Page Setup...**, etc. Be sure you use the reserved command number if you expect to get the default behavior from the THINK Class Library. For a list of all the reserved commands see the description for CBartender on page 191.

When you choose a command from the menu bar, the desktop sends a `FindCmdNumber` message to the bartender. The bartender matches the menu ID and the item number to a command number and sends it to the gopher in a `DoCommand` message. If the you use a Command key equivalent, the switchboard sends the `DoCommand` message to the gopher.

Note

Remember, the gopher is a pointer to a command object. Usually, the gopher points to a pane in the active window.

Using MENU resources

When you create your menus with ResEdit, append the command number to the menu item. The menu item and the command number are separated by the character **!**. For example, Figure 7-8 shows the **File** menu.

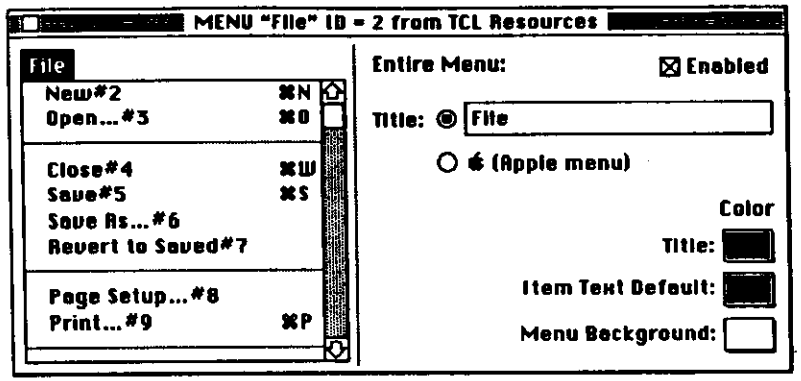

Figure 7-8 The File menu in ResEdit

Note

If you don't append a command number to a menu item, the bartender automatically assigns it `cmdNull`.

The `MENU` resources for these menus are in the file `TCL Resources`.

You can use any menu ID for your application's menus. The THINK Class Library reserves the following menu IDs for certain menus:

Menu title	Menu ID	Mnemonic
Apple	1	`MENUapple`
File	2	`MENUfile`
Edit	3	`MENUedit`
Font	10	`MENUfont`
Size	11	`MENUsize`

After you create all the menus that your application needs, create an `MBAR 1` resource that contains all of the IDs of the menus your application uses. The application's `SetUpMenus` message creates the bartender (stored in the global variable `gBartender`) to read the `MBAR 1` resource. The bartender creates the tables that match command numbers to menu items.

Note

Your application's menus must be in `MBAR 1` unless you change the defnition of `MBARapp` in `Constants.h`. or `TCL.p`.

If you want the bartender to return the menu ID and item number of a particular menu item, use the special command number –1 in your `MENU` resource. The bartender will return the negative of the menu ID in the high word and the menu item number in the low word. This is the same as menus that you build on the fly, described next.

Building menus on the fly

Menus that you create as your program is running, like Font menus, won't have command numbers associated with them. In this case, the bartender's `FindCmdNumber` method returns the negative of the menu ID in the high word and the item ID in the low word. When your `DoCommand` method gets a negative command number, you know you have to figure out the command from the menu ID and item number.

For example, if `DoCommand` gets a command -655369 (`0xFFF5FFF7`), it means that there is no command number associated with the menu item. To get the menu ID and the menu item, negate the value and split it into two words. In this example, the return value becomes 655369 (`0x000A0009`), which means that the menu ID is 10 and the menu item is 9.

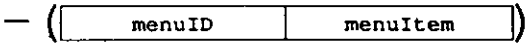

Figure 7-9 How FindCmdNumber builds command numbers.

You can add menu items to existing menus if you like. For example, you might want to add the Font menu to a general text-handling menu, or you might want to have a menu with the names of all the documents your application has opened. The important thing to remember is to add all these menus at the **end** of the existing menu. Otherwise, the bartender will get confused.

Dimming and checking menu items

The bartender includes methods to let you enable and disable and check and uncheck menu items. When you click in the menu bar, the bartender sends an `UpdateMenus` message to all of the bureaucrats in the chain of command. In the general case, all the items in the menu start out dimmed and unchecked. Then each bureaucrat enables the menu items that pertain

to it. Once the appropriate items have been enabled and checked, the Toolbox routine `MenuSelect` displays all the menus.

Note

The dimming, undimming, and unchecking take very little time. You won't notice a delay between the time you click on the menu bar and when the menu is displayed.

Suppose you click in the menu bar of a text processing application. When you click on the menu bar, but before the Toolbox displays the menu, the bartender disables all the menu items. Then, the application enables all the application related menu items: **New**, **Open...**, **Quit**, for example. The document enables all the document related items: **Save**, **Save As...**, **Revert** (if the document's been changed), and so on. A pane might check the current font and size in the **Font** menu. Finally the menu appears on the screen with the correct items checked and enabled.

The `UpdateMenus` method of your application, document, and pane need to enable each item. To make sure that item enabling happens from the general (application) to the specific (pane), be sure to call the inherited method first in your own `UpdateMenus` method.

You can use the bureaucrat methods `SetDimOption` and `SetUnchecking` in your application `SetUpMenus` method to modify this behavior. `SetDimOption` lets you specify whether the bartender should dim all, some, or none of the items when you click on the menu bar. For Font menus, for instance, it doesn't make sense to dim all the font names only to reenable them again.

Dim Option	Meaning
`dimNONE`	Never dim any of the menu items.
`dimSOME`	Dim only the menu items that have command numbers associated with them.
`dimALL`	Dim all of the menu items. Each bureaucrat's `UpdateMenus` method must enable the items for the commands it handles. This is the default.

There are only two options for SetUnchecking. The bartender either unchecks all the items or it doesn't.

Unchecking option	Meaning
TRUE	Uncheck all the menu items at menu selection. Your UpdateMenus method should check the appropriate items. Set this option for menus like Font menus or Style menus.
FALSE	Don't uncheck any menu items at menu selection. This is the default since most menu items never need to be checked.

Handling Low Memory Situations

The term "rainy day fund" comes from the expression "saving for a rainy day." It means to save money in case of disaster.

The CApplication class provides several methods that deal with low memory situations. These methods use a memory reserve called the **rainy day fund**.

Note

For details about the methods and instance variables described in this section, see Chapter 11, "CApplication."

When you call IApplication, you specify how much memory your application should allocate for the rainy day fund. You also specify how many bytes of that fund should be available to satisfy Toolbox routine memory requests and how many bytes of that fund should be available to satisfy critical operation requests. These values are stored in the instance variables toolboxBalance and criticalBalance.

If the Macintosh Memory Manager gets a request for more memory than is available, it calls a grow zone function. In the THINK Class Library, the grow zone function sends the application a GrowMemory message.

The GrowMemory method tries several strategies to free memory in the heap. First, it sends the application a MemoryShortage message. Your application subclass should override this method to release memory that is not crucial to execution. For instance, your MemoryShortage method might might dispose of a buffer it no longer needs. If the MemoryShortage method wasn't able to release enough memory, GrowMemory starts using the rainy day fund.

GrowMemory looks first at the inCriticalOperation instance variable to relase memory from the rainy day fund:

If inCriticalOperation is	leave this much in reserve
TRUE	toolboxBalance
FALSE	criticalBalance

A critical operation is a temporary operation that takes place in a single Macintosh event, that might take a lot of memory, and that should never fail. For instance, saving a file is a critical operation. You can use the routine SetCriticalOperation to set the inCriticalOperation flag.

If GrowMemory still wasn't able to release enough memory and the canFail flag is TRUE, GrowMemory returns without trying to allocate any more memory. If the canFail flag is TRUE, the application is saying that it can deal with a failing memory request.

If the canFail flag is FALSE, GrowMemory tries to release all the memory left in the rainy day fund because the application is not prepared to deal with a failing memory request. If there is not enough memory, even after using the rainy day fund, it is likely that the application will crash.

You can use the routine SetAllocation to set and reset the canFail flag. In general, your application should be prepared to handle failing memory requests.

Undoing and Mouse Tracking

The THINK Class Library provides a class that lets you implement the **Undo** command easily. In the default implementation, each document has its own undo history.

Undoing

The THINK Class Library uses the abstract class CTask to implement undoable actions. For every undoable action you want in your application, you need to create a subclass of CTask.

After you perform an action, you store enough information to undo it in your task's instance variables. Then you send the task to your supervisor in a Notify command. The CDocument class implements the Notify method to store a task in one of the document's instance variables. When you choose **Undo** from the **Edit** menu, the document's DoCommand method sends an Undo message to the task it stored.

Here's an example. Suppose you've defined a subclass of CTask to change the font in an edit text pane. Before passing the command on to the edit pane's DoCommand method, you create a task and store the current font in

an instance variable. After you pass the font command to the edit text pane, you send the task in a `Notify` message to the document.

Your `Undo` method would simply send the font change command to the document. Since the command goes through the regular command chain, your `DoCommand` method would create a task to let you undo what you were undoing.

Mouse tracking

The THINK Class Library uses the undo mechanism to make mouse tracking easier and undoable. The CMouseTask class is an abstract class that defines some methods specifically for mouse tracking.

To implement a mouse tracking task, define a subclass of CMouseTask and override the `KeepTracking` and `EndTracking` methods. The `KeepTracking` method does whatever you want to happen while the mouse is down. The `EndTracking` method does whatever you want to happen when the mouse is released.

For example, if you're moving a rectangle from one place in a pane to another, the `KeepTracking` method might draw a gray outline that moves as you move the mouse. The `EndTracking` method would erase the rectangle from its old location and redraw it in the new location.

If you want to make your mouse task undoable, you need to store enough information in the object to undo the effects of mouse tracking. You must also override the `Undo` method (inherited from CTask) to use this information to undo the effects of the mouse task.

After tracking the mouse, you can send the task in a `Notify` message to the document. When you choose **Undo** from the **Edit** menu, the document sends an `Undo` message to the task to undo the effects of mouse tracking.

Segmentation and the THINK Class Library

You can segment your application any way you want. The only thing to keep in mind is that certain files and libraries must be in a resident segment, that is, a segment that is never purged.

In THINK C, these files must be in a resident segment:

- `MacTraps`
- `MacTraps2`
- `oops (or oopsDebug)`
- `TCLUtilities.c`
- `LongCoordinates.c`
- `Exceptions.c`

See Chapter 7 of your THINK Pascal User Manual for more information about segmentation in THINK Pascal.

In THINK Pascal the following files must be in a resident segment. You can set the attributes of their segment to be Locked.

- `TCL.p`
- `Exceptions.p`
- `TCLRuntime.lib`
- `Interface.lib`
- `TCL.lib`

Note

The directive segments «`%_MethTables`» and «`%_SelProcs`» must also be in a resident segment. If you are using Far Code, «`%_MethTables`» must be in a segment by itself.

Debugging and the THINK Class Library

The THINK C and THINK Pascal versions of the THINK Class Library use some compiler-specific features to help you debug programs that use the TCL.

Debugging aids in THINK C

When you're debugging your application, you should use the library oopsDebug. This library contains a version of the message dispatcher that checks for NULL objects or possibly missing methods. Though there's no problem in building an application with this library, it is slightly less efficient than the regular oops library.

If the preprocessor symbol `__TCL__DEBUG__` is defined, certain error-checking functions like the `ASSERT` macro are enabled. The `ASSERT` macro takes a Boolean expression as an argument an raises an exception if the assertion is FALSE.

In THINK C, the global variables `gBreakFailure` and `gAskFailure` let you examine and simulate exceptions. If `gBreakFailure` is TRUE, THINK

C calls the Toolbox routine Debugger when an exception is raised. If `gAskFailure` is TRUE, THINK C calls Debugger when the utility routines like `FailOSErr` that may raise exceptions are called. You can then change the arguments to the utility routine to simulate an exception.

There are two underscores at the beginning and end of `__TCL_DEBUG__`. You can set the value of `__TCL_DEBUG__` in the Prefix page of the **Options...** dialog.

Debugging aids in THINK Pascal

The THINK Pascal version of the THINK Class Library uses three compiler variable to control debugging. If the compiler variable `TCL_DEBUG` is TRUE, the `Assert` procedure is enabled. The `Assert` procedure evaluates a Boolean expression and raises an exception if the result is FALSE.

Note

In `Starter.π`, `TCL_DEBUG` is set to TRUE. In `Starter.Build.π` it's set to FALSE.

If the compiler variable `ResumeAfterError` is TRUE, any time an exception is raised, an alert box appears, and the program ends.

Another compiler variable, `DebugExceptions` is used only when `ResumeAfterError` is FALSE. If `DebugExceptions` is TRUE, the THINK Class Library tries to handle exceptions in the THINK Pascal environment. The debugging option must be on for any file that may raise an exception.

If `ResumeAfterError` is FALSE and `DebugExceptions` is FALSE, theTHINK Class Library handles exceptions in the normal way, but not in the THINK Pascal environment. These are the settings in `Starter.Build.π`.

Use the **Compile Options...** command to set these compiler variables.

THINK Class Library Resources

The THINK Class Library requires certain resources in your project's resource file. All of the resources described in this section are in the file `TCL Resources`. The mnemonic constants for all of these resources are in the file `Constants.h` in the `Core Headers` folder in THINK C and in `TCL.p` in THINK Pascal.

Alerts

The `ALRT` and `DITL` resources always have matching IDs. You can change these resources to suit your application.

ALRT/DITL	Used for
128	General. A handy, all-purpose alert box. The `DITL` contains only "^0", so you can use the Toolbox routine `ParamText` to set up the text.
150	Confirm to revert to last saved version.
151	Confirm to save changes before closing or quitting.
200	Severe Macintosh error has occurred.
250	No printer selected.
251	Error alert. The DITL says "Couldn't complete the last command because ^0."
252	Error alert. The DITL says "Couldn't successfully startup or quit the application because ^0"
253	Assertion failed. Used by assertion in exception handling.
300	Macintosh OS error alert.

Controls

The THINK Class Library uses this `CNTL` template for all the scroll bars it creates.

CNTL	Used for
300	Scroll bar

Error message strings

The THINK Class Library uses a resource of type `Estr` to report Macintosh errors. `Estr` resources have exactly the same format as `STR` resources. You can use the ResEdit command **Open as Template...** in the **File** menu to open and edit an `Estr` resource as a `STR`.

The ID of the `Estr` resource is the error code you want to identify. The file `TCL Resource` includes one `Estr`. The error handling class CError uses `Estr` resources to display messages. You should create an `Estr` for every error your application reports to the user.

Estr	Used for
-108	Out of memory
-192	Tried to get nonexistent resource

Menus

The THINK Class Library reserves these menu IDs for the standard menus. The **File** and **Edit** menus contain all the standard items. You can remove the

ones that don't apply to your application. The bartender builds the desk accessory menu for you automatically, but you'll have to build the **Font** and **Size** menus yourself in the `SetUpMenus` method of your application. For an example of Font and Size menus, see the `TinyEdit` project in the `TCL 1.1 Demos` folder.

MENU	Used for
1	Apple
2	File
3	Edit
10	Font
11	Size

Note

In THINK C the mnemonics for these menus are in `Commands.h`, not in `Constants.h`. In THINK Pascal, these constants are in `TCL.p`.

Menu bars

The THINK Class Library uses this resource to install all the menus in your application. The `MBAR` resource in `TCL Resources` automatically includes the **Apple**, **File**, and **Edit** menus.

MBAR	Used for
1	List of all menus to install at application startup.

Small icon

Earlier versions of THINK Class Library used this small icon to draw a grow box instead of the Toolbox routine `DrawGrowIcon`. If your application uses `SICN`s, you can use the routine `DrawSICN` to draw it in a pane. See page 485 for details.

SICN	Used for
200	Grow box

Strings and string lists

The THINK Class library uses these strings for various prompts and messages. You can modify these to suit your application.

STR	Used for
150	Prompt for the **Save As...** dialog box.
300	Generic operating system error message used when no `Estr` resource is available.
301	Generic error suffix. "of a Mac OS Error"

STR#	Used for
128	List of common Macintosh words. This list includes the words: quitting, closing, Undo, Redo, Untitled, Show Clipboard, Hide Clipboard.
129	Strings used for low memory warnings.
130	Task names for changing the wording of the Undo menu item text. This resource has no strings. Your application should add strings to this list if it supports Undo. See the descriptions of CTask and CMouseTask.
131	Strings used by exception handler.

Window template

The THINK Class Library requires only one window template for the Clipboard window. Your application will probably define one or more additional `WIND` templates.

WIND	Used for
200	Clipboard. Window template used for displaying the clipboard.

Modifying the THINK Class Library

You can modify the THINK Class Library classes to suit your particular needs. To change the behavior of one of the THINK Class Library classes, create a new subclass of the class you want to modify, and add new methods or override the methods you need to change. In some cases, it might be a better idea to change the source code for a class.

Be aware that there are dangers in changing the source code of a class. From time to time, Symantec may release new versions of the THINK Class Library. Changes that you make in the source of a class may make it difficult to use new classes or updates of existing classes. If you do make changes to the source of a class, be sure to keep an archival copy of the original class and to mark your changes clearly.

The second danger is a little more subtle. As you use the THINK Class Library, you may want to create new classes and subclasses to implement some kind of behavior. If you want to be able to use these classes in other programs, and especially if you want others to be able to use these classes, you should not rely on any features of your modified classes.

Note

Under your license agreement, you may distribute new subclasses of the classes in the THINK Class Library. You may not, however, distribute modified sources of the classes in the THINK Class Library.

Where to Go Next

Learning to use the THINK Class Library takes time and experimentation. Start with the `TinyEdit` example in the `TinyEdit Folder` It was built from the Starter application you should use to create your own applications. You might start by adding an **About...** box to the `TinyEdit` application. (Hint: Since it's an application-wide command, implement it in the application's `DoCommand` method.) As you explore how the `TinyEdit` application was put together, look at the next chapters to understand how the classes of the THINK Class Library work.

Exception Handling 8

The THINK Class Library includes a new exception handling mechanism to detect and respond to failures. The exception handling mechanism lets you divide your routines into two parts: one to handle the normal operation and another to handle abnormal situations.

Note

Although the exception handling routines are designed to work with the THINK Class Library, you can use it in a limited way with non-TCL programs.

Contents

What Is an Exception Handler? . . . 129

Using the Exception Mechanism . . . 130
- Exception handling conventions . . . 131
- Displaying error messages . . . 131

Exception Handling in THINK C . . . 132
- The basic form . . . 132
- Use handlers only when necessary . . . 133
- Returning values . . . 134
- Special cases . . . 135

Exception Handling in THINK Pascal . . . 135
- The basic form . . . 135
- Use handlers only when necessary . . . 137
- Using Exit with exception handlers . . . 139
- Special cases . . . 139

Exception Handler Routines . . . 139
- Exception handler routines . . . 139
- Exception raising routine . . . 140
- Error detection routines . . . 141
- Utility routines . . . 141

What Is an Exception Handler?

An exception is the occurrence of an abnormal condition during the execution of a program. For example, your program may open a file, write and read data to it, and close it. That's the normal operation for your program. In the course of operation, though, your program might encounter several exceptions: the file may not exist, the file may be unavailable, there might be media errors, or your program may not be able to close the file.

The exception handling mechanism in the THINK Class Library lets you separate the normal execution part of a routine from the error handling part of the routine. Without an exception handler, your program might have this kind of structure:

```
create a file object
if creation failed
   report an error
else
   open the file
   if open failed
      delete the file object
      report an error
   else
      allocate a buffer
      if allocation failed
         delete the file object
         report an error
      else
         ...
```

As you can see, the normal flow of control is hidden in all the error-checking code. With an exception handler, your code looks more like this:

```
TRY TO DO THE FOLLOWING:
   create a file object
   open the file
   allocate a buffer
   read the contents of the file
   close the file
CATCH ANYTHING THAT WENT WRONG:
   if the file object was created, delete it
   if the buffer was allocated, release it
   if the file is still open, close it
```

The normal portion is called the **try handler**, and the portion that handles abnormal conditions is called the **catch handler**. A routine in the try block can **raise an exception** to stop normal operation and go to a catch block.

The routine that raises an exception is called `Failure`. The exception handling mechanism has several utility routines that test their parameters and call `Failure` for you if an error actually occurred. These routines are described later on page 141.

The exception handling mechanism keeps a stack of catch handlers. If a routine raises an exception, and there is no catch handler in the routine, control passes to the most recent catch handler. The THINK Class Library has a catch handler in CApplication's Run method, so any exception is always caught.

Suppose you have three functions, and each with a catch and try handler:

```
function A
   TRY
      call B
   CATCH
      clean up A
end

function B
   TRY
      call C
   CATCH
      clean up B
end

function C
   TRY
      do something
   CATCH
      clean up C
end
```

If an operation in function C's try handler raises an exception, control passes to C's catch handler. When that handler is finished, control passes to B's catch handler, and then finally to A's catch handler. If B had no catch handler, control would go from C's catch handler to A's catch handler.

Not every routine needs to have try and catch handlers, just those that would need to clean up if an error had occurred. Some of the time, you can rely on a caller's catch handler to do the cleanup for you.

Using the Exception Mechanism

The exception handling mechanism is conceptually the same in both THINK C and in THINK Pascal, but, because of differences in the languages, you use them differently. In THINK C, the exception handler uses C macros to divide the normal portion of your functions from the exception portions. In THINK Pascal, it uses nested procedures.

This section discusses some aspects of exception handlers common to both languages. The next two sections cover specific aspects of the exception handler for THINK C and for THINK Pascal.

Exception handling conventions

Regardless of the language you're using, you should follow certain conventions for writing try and catch handlers.

If an object cannot be created (that is, if a call to `new` fails) the THINK Class Library raises an exception with `FailNIL`. In THINK C, CObject's `operator new` is overridden. In THINK Pascal, you must use `TCLRuntime.lib`.

Once you've created an object successfully, you need to make sure that it will be released if the initialization fails. You can set up a catch handler to dispose of the object, or, if the object reference is an instance variable of another class, you can rely on its catch handler to dispose of the object. The examples below show you how to do this.

Make sure that you set all object references to NIL before you create them with new. This way you can tell whether the object was created because the object reference will be non-NIL.

The memory management methods are designed to protect your application from running out of memory in extreme circumstances. In normal operation, you should check all of your memory allocations without relying on the memory reserves. See page 164 for more information about handling low memory situations.

Displaying error messages

Since catch handlers propagate, you can rely on the top-level catch handler in CApplication's Run method to display an error message. If you want to display your own message, you have to make sure that the catch handler in Run does not display a message.

See page 492 for a description of the `ErrorAlert` utility routine.

When a try handler wants to raise an exception, it calls `Failure`, which takes two arguments: the error that caused the failure and a message. These two values are saved in the globals `gLastError` and `gLastMessage`. The error is typically an operating system error. The message is optional. It's used by the `ErrorAlert` routine to find an error string to display.

Note

In Pascal, you can look at the parameters of the catch handler procedure to determine the error and message values.

You can display your own error message in your catch handler. If you do, be sure to call `SetFailInfo` with an error code `kSilentError`. This special error code tells the top-level handler not to display an error message at all. Whatever you pass to `SetFailInfo` as the message is ignored.

Exception Handling in THINK C

In THINK C, you use the `TRY`, `CATCH`, and `ENDTRY` macros to write exception handlers. The macros are defined in `Exceptions.h`.

The basic form

This is the basic form for writing an exception handler in C:

```
void Example(void)
{
    TRY {
        This is the normal flow.
        Anything here can raise an exception that will be
        caught in the catch handler.
    }
    CATCH {
        This is the catch handler.
        Clean up after errors here: close files, release memory, etc.
    }
    ENDTRY;
}
```

Here's a concrete example:

```
void Example (void)
{
    CThing *anObject = NULL;

    TRY {
        anObject = new CThing;
        anObject->IThing();
    }
    CATCH {
        ForgetObject(anObject);
    }
    ENDTRY;
}
```

Example 8-1 A simple exception handler in C

In Example 8-1, there are two places where an exception can be raised. First, when you create `anObject`, there may not be enough memory. If that's the case, new fails and raises an exception. Second, an operation in the `IThing` initialization method may raise an exception. In either case, control passes to the catch handler.

If new failed, `anObject` will be NULL, so the call `ForgetObject` won't do anything, and there's no other cleanup to do. If new succeeded, then the exception was caused by some operation in `IThing`. In that case, `ForgetObject` sends a `Dispose` message to `anObject` to release it.

Use handlers only when necessary

You don't need a catch handler every time you create an object if you know that it will be disposed of by another handler. Here's an example of that:

```
void CMyApp::CreateDocument(void)
{
    CMyDoc *newDoc = NULL;

    TRY {
        newDoc = new CMyDoc;
        newDoc->IMyDoc(...);
        newDoc->NewFile();
    }
    CATCH {
        ForgetObject(newDoc);
    }
    ENDTRY;
}

void CMyDoc::NewFile(void)
{
    BuildWindow();
    itsWindow->Select();
}

void CMyDoc::BuildWindow()
{
    CCoolPane *myPane;

    itsWindow = new CWindow;
    itsWindow->IWindow(...);

    myPane = new CCoolPane;
    myPane->ICoolPane(...);
    ...
}
```

Example 8-2 A set of routines with only one catch handler in C

In Example 8-2 only `CMyApp::CreateDocument` needs a catch handler. Here's why. `BuildWindow` creates a window and immediately stores into `itsWindow`, which is an instance variable in CMyDoc. `BuildWindow` also creates a pane, which is immediately added into the view hierarchy by `ICoolPane`. If an exception is raised, it will be caught in `CreateDocument`'s catch handler, which calls `ForgetObject`. `ForgetObject` sends a `Dispose` message to `newDoc`, which sends a `Dispose` message to `itsWindow`, which in turn disposes of all the panes that the window encloses.

Returning values

If your function returns a value, you must not return from the try handler. Instead, your function should return after `ENDTRY`. For example, look at `CDataFile::ReadAll`:

```
Handle CDataFile::ReadAll (void)
{
    register OSErr errCode;
    long length;
    Handle contents = NULL;

    TRY
    {
        FailOSErr(GetEOF(refNum, &length));
        contents = NewHandleCanFail(length);
        FailNIL(contents);
        FailOSErr(SetFPos(refNum,
                          fsFromStart, 0L));
        FailOSErr(FSRead(refNum, &length,
                         *contents));
    }
    CATCH
    {
        ForgetHandle(contents);
    }
    ENDTRY;

    return contents;
}
```

Example 8-3 Returning from a function with try and catch handlers

The `ReadAll` method in Example 8-3 is a good example of using the exception handling mechanism. Note how all of the File Manager routines that return error codes are wrapped in calls to `FailOSErr`. This utility function checks the error return value, and if it is not `noErr`, it raises an exception. Note also how the function uses `NewHandleCanFail` to allocate a handle. That function ensures that the memory manager won't use any of the memory reserves to get memory. The call to `FailNIL` raises an exception if the handle couldn't be allocated.

Special cases

In some abnormal cases, you may want to keep catch handlers from propagating to the top-level handler in `CApplication::Run`. If so, just use the macro `NO_PROPAGATE` somewhere in your catch handler.

If you think you can fix whatever raised the exception in your catch handler, you can use the `RETRY` macro to retry the try handler:

```
void Example(void)
{
    TRY {
        something that might raise an exception
    }
    CATCH {
        fix the problem
        RETRY;
    }
    ENDTRY;
}
```

Example 8-4 Using `RETRY`

Exception Handling in THINK Pascal

In THINK Pascal, you use a nested procedure to write an exception handler. Before the main flow of control, you use the `CatchFailures` procedure to set up the nested procedure as the exception handler. Your routine must also declare a variable of type `FailInfo` to hold the information the exception handling mechanism needs.

Warning

For the exception handler to work correctly under the THINK Pascal environment, you must have debugging enabled for all procedures and functions that may raise an exception. The exception mechanism works correctly when you build an application.

The basic form

This is the basic form for writing an exception handler in Pascal:

```
procedure Example;
    var
        fi: FailInfo; { info for exception handler }
```

```
    procedure HandleFailure (error: integer;
                             message: longint);
    begin
        This is the catch handler.
        Clean up after errors here:
        close files, release memory, etc.
    end;

begin
    CatchFailures(fi, HandleFailure);

    This is the normal flow.
    Anything here can raise
    an exception that will be
    caught in the catch handler.
    Success;
end;
```

Note that you must declare a variable of type `FailInfo` in the main procedure. By convention, this variable is called `fi`. To set up the exception handler, you call the procedure `CatchFailures` and pass it the `fi` variable and the name of the nested procedure that serves as the catch handler. Finally, you must call `Success` to let the exception handler know that the main flow of code executed successfully.

The catch handler is always a nested procedure declared like this:

```
procedure HandleFailure (error: integer;
   message: longint);
```

`Error` and `message` are the values that were passed to the `Failure` routine to raise the exception. `Typically`, error is a Macintosh error code, and `message` is an optional application-specific description of what went wrong.

Here's a concrete example:

```
procedure Example;
   var
      anObject: CThing;
      fi: FailInfo;

   procedure HandleFailure (error: integer;
                            message: longint);
   begin
      ForgetObject(anObject);
   end;

begin
   anObject := NIL;

   CatchFailures(fi, HandleFailure);

   new(anObject);
   anObject.IThing;

   Success'
end;
```

Example 8-5 A simple exception handler in Pascal

In Example 8-1, there are two places where an exception can be raised. First, when you create `anObject`, there may not be enough memory. If that's the case, new fails and returns NIL. Second, an operation in the `IThing` initialization method may raise an exception. In either case, control passes to the catch handler.

If new failed, `anObject` will be NIL, so the call `ForgetObject` won't do anything, and there's no other cleanup to do. If new succeeded, then the exception was caused by some operation in `IThing`. In that case, `ForgetObject` sends a `Dispose` message to `anObject` to release it.

Use handlers only when necessary

You don't need a catch handler every time you create an object if you can be sure that it will be disposed of properly by another handler. Here's an example of that:

```
procedure CMyApp.CreateDocument;
begin
   var
      newDoc: CMyDoc;
      fi: FailInfo;

   procedure HandleFailure (error: integer;
```

```
                                        message: longint);
    begin
       ForgetObject(newDoc);
    end;

begin
    newDoc := NIL;

    CatchFailures(fi, HandleFailure);

    new(newDoc);
    newDoc.IMyDoc(...);
    newDoc.NewFile;

    Success;
end;

procedure CMyDoc.NewFile;
begin
    BuildWindow;
    itsWindow.Select;
end;

procedure CMyDoc.BuildWindow;
    var
       myPane: CCoolPane;
       tmpWin: CWindow;
begin

    new(tmpWin);
    itsWindow := tmpWindow;
    itsWindow.IWindow(...);

    new(myPane);
    myPane.ICoolPane(...);
    ...
end;
```

Example 8-6 A set of routines with only one catch handler in Pascal

In Example 8-2 only `CMyApp.CreateDocument` needs a catch handler. Here's why. `BuildWindow` creates a window and immediately stores into `itsWindow`, which is an instance variable in CMyDoc. `BuildWindow` also creates a pane, which is immediately added into the view hierarchy by `ICoolPane`. If an exception is raised, it will be caught in `CreateDocument`'s catch handler, which calls `ForgetObject`. `ForgetObject` sends a `Dispose` message to `newDoc`, which sends a `Dispose` message to `itsWindow`, which in turn disposes of all the panes that the window encloses.

Using Exit with exception handlers

If you use the Pascal keyword `Exit` to leave a function or procedure that has a catch handler, be sure that you call `Success` first. Otherwise, the exception mechanism will get confused.

Special cases

In some abnormal cases, you may want to keep catch handlers from propagating to the top-level handler in `CApplication.Run`. If so, you should insert a label after the call to `Success` in the try handler and goto it from your catch handler.

If you think you can fix whatever raised the exception in your catch handler, you can use the `RetryException` procedure to retry the try handler:

```
procedure Example;
var
    fi: FailInfo;

    procedure HandleFailure (error: integer;
                                    message: longint);
    begin
       fix the problem
       RetryException(fi);
    end;

begin
    something that might raise an exception
end;
```

Example 8-7 Retrying the try handler

Exception Handler Routines

These routines form the exception handling mechanism. Your program should use the error detection routines to test the return values of Macintosh Toolbox routines.

Exception handler routines

If you're a Pascal programmer, you should only use `CatchFailures`, `Success`, and `RetryException`. If you're a C programmer, the `TRY`, `CATCH`, and `ENTRY` macros call these routines for you.

CatchFailures

```
procedure CatchFailures (var fi: FailInfo;
   procedure handler (error: integer;
                      message: longint));
```

Pascal only

This procedure sets up a catch handler. `Fi` is a variable of type `FailInfo` declared in the outer procedure or function. `Handler` is the name of a nested procedure that will serve as the catch handler.The catch handler is always a nested procedure declared like this:

```
procedure HandleFailure (error: integer;
   message: longint);
```

`Error` and `message` are the values that were passed to the `Failure` routine to raise the exception. `Typically`, error is a Macintosh error code, and `message` is an optional application-specific description of what went wrong.

PushTryHandler

C only

```
void PushTryHandler (FailInfo *fi);
```

Push an exception handler on the exception stack. This function is called by the TRY macro. You should not need to call this function yourself.

Success

```
procedure Success;
void Success (void);
```

Pop the most recent exception handler off the exception stack. In THINK Pascal, you should call this procedure after all the code that could have failed has executed successfully. In THINK C, the CATCH macro calls this function for you.

RetryException

```
procedure RetryException (var fi: FailInfo);
void RetryException (FailInfo *fi);
```

Retry the try handler. In THINK Pascal you can call this procedure if you can fix the problem in the catch handler. In THINK C, you should use the RETRY macro in the catch handler.

Exception raising routine

Failure

```
procedure Failure (error: integer; message: longint);
void Failure (short error, long message);
```

Raise an exception. Set `gLastError` to error and `gLastMessage` to message. If `error` is not `kSilentError`, the catch handler in CApplication's Run method displays an error alert.

For an explanation of the message parameter, see the description of `ErrorAlert` on page 492

Error detection routines

These routines test their arguments or check Toolbox error-reporting functions to raise and exception if an error has occurred. You should use these routines any time you call any Toolbox routines that returns an error code, that deals with resources, or moves memory.

FailMemError

```
procedure FailMemError;
void FailMemError (void);
```

Raise an exception if the Toolbox function `MemError` returns anything but `noErr`. Calls `Failure(memoryError, 0)`.

FailNIL

```
procedure FailNIL (p: UNIV Ptr);
void FailNIL (void *p);
```

Raise an exception if p is NIL. Calls `Failure(memFullError, 0)`.

FailResError

```
procedure FailResError;
void FailResError (void);
```

Raise an exception if the Toolbox function `ResError` returns anything but `noErr`. Calls `Failure(resourceError, 0)`.

FailNILRes

```
procedure FailNILRes (resH: UNIV Handle);
void FailMemError (void resH);
```

Raise an exception is resH, a resource handle, is NIL. If `ResError` returns `noErr`, call `Failure(resNotFound, 0)`. Otherwise, calls `Failure(resourceError, 0)`.

FailOSErr

```
procedure FailOSErr (errCode: OSErr);
void FailOSErr (OSErr errCode);
```

Raise an exception if `errCode` is anything but `noErr`. This function calls `Failure(errCode, 0)`.

Utility routines

SpecifyMsg

```
function SpecifyMsg (strListID:integer;
   strIndex: integer): longint;
long SpecifyMsg (short strListID, short strIndex);
```

Create a message value. `StrListID` is the resource ID of a `STR#` resource, and `strIndex` is the index into that resource. The value returned should be

passed to `Failure` (or one of the routines that calls `Failure`). See the description of `ErrorAlert` on page 492 for more information.

SetFailInfo

```
procedure SetFailInfo (newError: integer;
   newMessage: longint);
void SetFailInfo (short newError, long newMessage);
```

Set the global variables gLastError and gLastMessage. If gLastMessage is not zero, this routine doesn't reset it. GLastError is always set.

Although it's usually used internally by the exception handling mechanism, you should call `SetFailInfo(kSilentError, 0)` to prevent the top-level exception handler from displaying an error message.

CAbstractText 9

Introduction

CAbstractText is an abstract class that provides you with a template for creating a pane that displays text.

Heritage

Superclass	CPanorama
Subclasses	CEditText

Using CAbstractText

CAbstractText is an abstract class for creating panes that display text. A text pane is usually the panorama in a scroll pane so you can scroll through the text. The `Specify` method, on page 146, lets you choose whether the user can edit and copy your text pane's text. The `DoCommand` method, on page 148, handles all the common text editing commands such as cutting and pasting, font selection, line spacing, etc.

To handle the **Undo** command, CAbstractText creates task objects for many common commands and sends messages to the object telling it to perform or undo its task. CTextEditTask on page 447 handles typing, cutting, copying, and pasting. CTextStyleTask on page 457 handles font, size, and style commands.

When you perform a task that can be undone, the name of the **Undo** commaond changes, for example **Undo Typing**. The **Undo** labels for this CAbstractText are stored in the resource STR# 130. The index of the first string is stored in the class variable `cFirstTaskIndex`. By default, it's 1. If you put the strings in a different place in STR# 130, you should set this variable to the index of the first string. If you don't use **Undo** labels, set this variable to 0. The labels must be in this order: Typing, Cut, Copy, Paste, Clear, Formatting.

Since it's an abstract class, you can't create an instance of CAbstractText. You can use the CEditText or CStyleText classes that come with the THINK Class Library.

These classes are based on the Macintosh TextEdit (TE) routines and become slow if they contain over 4000 characters. If your text pane needs to handle more text, create your own subclass of CAbstractText.

Most of CAbstractText's methods do nothing. They are "place holders" for the methods your subclass must override. The rest of CAbstractText's methods call them. These are the methods you must override:

`DoClick`	`SetTextPtr`
`GetTextHandle`	`SetFontNumber`
`SetFontStyle`	`SetFontSize`
`SetTextMode`	`SetAlignCmd`
`SetAlignCmd`	`SetSpacingCmd`
`GetHeight`	`GetCharOffset`
`GetCharPoint`	`GetTextStyle`
`GetCharStyle`	`FindLine`
`GetLength`	`TypeChar`
`SetSelection`	`GetSpacingCmd`
`GetSelection`	`GetNumLines`
`CopyTextRange`	`PerformEditCommand`
`InsertTextPtr`	`Dawdle`

These two methods assume that the text in your pane is stored in a single contiguous buffer. If it's not, you should override them:

`GetCharBefore`	`GetCharAfter`

Variables

This is an instance variable that any class can use.

Variable	Type	Description
`itsTypingTask`	`CTextEditTask`	Active typing task.

`CFirstTaskIndex` is a class variable that only CAbstractText and its subclasses should use. In THINK C, it's a class variable.

Variable	Type	Description
`cFirstTaskIndex`	`short`	Index in `STR#` 130 resource of first undo label

These are internal instance variables which only subclasses of CAbstractText should use. In THINK C, they are protected.

Variable	Type	Description
`lineWidth`	`integer`	Width of a text line in pixels. If negative, lines are as wide as the pane.
`fixedLineHeights`	`Boolean`	TRUE if all lines same height.
`wholeLines`	`Boolean`	TRUE if lines aren't cut off vertically.
`lastFontNum`	`integer`	Last font number.
`lastFontCmd`	`longint`	Last font command number.
`lastTextSize`	`integer`	Last seen text size
`lastSizeCmd`	`longint`	Last size command number.
`editable`	`Boolean`	TRUE if user can edit text.
`stylable`	`Boolean`	TRUE if user can change font, size, or style of text.

Methods

Construction and destruction methods

IAbstractText

```
procedure IAbstractText (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption;
   aLineWidth: integer);
void IAbstractText (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth);
```

Initialize an abstract text pane. Most of the arguments to this method are identical to pane initialization. `ALineWidth` specifies how wide the lines should be. If it's less than zero, the text wraps to the width of the pane.

Note

The descriptions of the other arguments are in CPane.

IViewRes

```
procedure IViewRes (rType:ResType; resID: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize edit text pane from a resource template. `RType` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the enclosure and supervisor of the pane.

To initialize a text pane from a resource file, use a `'AbTx'` resource.

Note

Earlier versions of the THINK Class Library used `'StTx'` resources to initialize static text and edit text panes. You should make sure that any old programs do not use `'StTx'`.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of an abstract text object.

Specify

Accessing method

```
procedure Specify (fEditable, fSelectable, fStylable:
   Boolean);
void Specify (Boolean fEditable,
   Boolean fSelectable, Boolean fStylable);
```

Specify whether the user can edit, select, and style text. This method sets `editable` to `fEditable`, `wantsClicks` to `fSelectable`, and `stylable` to `fStylable`. By default, text is editable, selectable, and stylable. If editing is allowed, you can insert and delete text. If selecting is allowed, you can select text to copy it to the clipboard. If styling is allowed, the user can change have change the font, style, or size of the text.

To make your `Specify` calls more understandable, the THINK Class Library defines some constants you can use instead of TRUE and FALSE:

`kEditable`	`kNotEditable`
`kSelectable`	`kNotSelectable`
`kStylable`	`kNotStylable`.

This list describes some common ways to call `Specify`:

- To let the user edit and style text, call `Specify(kEditable, kSelectable, kStylable)`.
- To prevent the user from editing, styling, or selecting text, call `Specify(kNotEditable, kNotSelectable, kNotStylable)`.
- To let the user select text (to copy it, for example) but not edit or style it, call `Specify(kNotEditable, kSelectable, kNotStylable)`
- To let the user edit text, but not style it, call `Specify(kEditable, kSelectable, kNotStylable)`.
- To let the user style text, but not edit it, call `Specify(kNotEditable, kSelectable, kStylable)`

Note

If `fSelectable` is `kNotSelectable`, `Specify` disables editing and styling , regardless of the values of `fEditable` and `fStylable`.

You can call `Specify` at any time. For example you could lock text (that is, disable editing) to prevent accidental editing, then let the user unlock it later to edit it.

GetSpecification

```
procedure GetSpecification (var fEditable,
   fSelectable, fStylable: Boolean);
void GetSpecification (Boolean *fEditable;
   Boolean *fSelectable, Boolean *fStylable);
```

Return whether this pane is editable, stylable, and selectable.

Command methods

DoCommand

```
procedure DoCommand (theCommand: longint);
void DoCommand (long theCommand);
```

This method handles the common commands that apply to text: cutting and pasting, font selection, styling, alignment, and spacing. DoCommand handles these commands: (It passes all other commands to its supervisor.)

- Editing commands: cmdCut, cmdCopy, cmdPaste, cmdClear, cmdSelectAll
- Style commands: cmdPlain, cmdBold, cmdItalic, cmdUnderline, cmdOutline, cmdShadow, cmdCondense, cmdExtend
- Alignment commands: cmdAlignLeft, cmdAlignCenter, cmdAlignRight
- Spacing commands: cmdSingleSpace, cmd1HalfSpace, cmdDoubleSpace.
- Font selection from the **Font** menu (MENUfont)
- Size selection from the **Size** menu (MENUsize)

To let the user undo actions, this method creates a task object for most commands. It creates a CTextEditTask object for an editing command and a CTextStyleTask object for a style, alignment, spacing, font, or size command.

UpdateMenus

```
procedure UpdateMenus;
void UpdateMenus (void);
```

Update the menus that have to do with text processing right before the menu appears. This method enables the **Cut**, **Copy**, and **Clear** commands if there is a current selection and enables the **Paste** command if there is text in the clipboard. This menu also checks the current font, size, style, alignment, and spacing commands.

Note

See the implementation of this method if you want an example of how to write your own UpdateMenus method.

DoKeyDown

```
procedure DoKeyDown (theChar: char; keyCode: Byte;
   macEvent: EventRecord);
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handle a key down in a text pane. Usually, this passes function and command keys to its superclass and inserts other characters into the text. This table shows how it handles command keys and function keys:

If the key is...	This method...
Any command key	Passes it to the superclass.
Home, Page Up, Page Down	Passes it to the superclass.
End	Scrolls to the bottom left of the text.
A cursor key	Calls TypeChar and notifies the editing task of the change in the selection.
Any other key	Passes it to the editing task, which should insert the character if the text pane is editable.

DoAutoKey

```
procedure DoAutoKey (theChar: char; keyCode: Byte;
   macEvent: EventRecord);
void DoAutoKey (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handle an auto-key in a text pane. If the command key is down, this method does nothing. Otherwise, this method calls DoKeyDown.

PerformEditCommand

```
procedure PerformEditCommand (theCommand: longint);
void PerformEditCommand (long theCommand);
```

Perform the standard cut, copy, paste, and clear commands on the text. The task classes call this method to perform and undo an edit command. The default method does nothing. Your subclass must override this method.

TypeChar

```
procedure TypeChar (theChar: char;
   theModifiers: integer);
void TypeChar (char theChar, short theModifiers);
```

Process a keystroke. This method does not need to set up for an **Undo** command and should handle the key directly. The task classes call this method to perform a key stroke. The default method does nothing. Your subclass must override this method.

SelectionChanged

```
procedure SelectionChanged;
void SelectionChanged (void);
```

The selection has just changed. This method sends a `BroadcastChange` message with `textSelectionChanged` as the reason. If this text pane has an editing task, it sends the task a `SelectionChanged` message.

MakeEditTask

```
function MakeEditTask (editCmd: longint):
   CTextEditTask;
CTextEditTask *MakeEditTask (long editCmd);
```

Create a task to handle typing and the cut, copy, paste, and clear commands. This method creates a CTextEditTask class. If you override this method, it must create a subclass of CTextEditTask. In THINK C `MakeEditTask` is a protected method.

MakeStyleTask

```
function MakeStyleTask (styleCmd: longint):
   CTextStyleTask;
CTextStyleTask *MakeStyleTask (long editCmd);
```

Create a task to handle font, style, size, alignment, and spacing commands. This method creates a CTextStyleTask class. If you override this method, it must create a subclass of CTextStyleTask. In THINK C `MakeStyleTask` is a protected method.

BecomeGopher

```
procedure BecomeGopher (fBecoming: Boolean);
void BecomeGopher (Boolean fBecoming);
```

If `fBecoming` is TRUE, this text pane is becoming the gopher. This method activates it and enables the **Font**, **Size**, and **Style** menus. Otherwise, this text pane is no longer the gopher. This method deactivates the pane and disables the **Font**, **Size**, and **Style** menus.

Display methods

ScrollToSelection

```
procedure ScrollToSelection;
void ScrollToSelection (void);
```

Scroll so the current selection is visible.

SetSelection

```
procedure SetSelection (selStart, selEnd: long;
   fRedraw: Boolean);
void SetSelection (long selStart, long selEnd,
   Boolean fRedraw);
```

Set the selection to the range corresponding to character positions `selStart` through `selEnd`. The default method does nothing. Your subclass must override this method.

GetSelection

```
procedure GetSelection (var selStart, selEnd:
   longint);
void GetSelection (long *selStart, long *selEnd);
```

Return the start and end of the current selection. The default method does nothing. Your subclass must override this method.

SelectAll

```
procedure SelectAll (fRedraw: Boolean);
void SelectAll (Boolean fRedraw);
```

Select all the text in this text pane. This method uses `GetLength` to determine the length of the text.

Paginate

```
procedure Paginate (aPrinter CPrinter;
   pageWidth, pageHeight: integer);
void Paginate (struct CPrinter *aPrinter,
   short pageWidth, short pageHeight);
```

Split the text pane into pages. This method makes sure that lines aren't cut in half horizontally at the ends of pages.

Text specification methods

SetTextString

```
procedure SetTextString (textStr: Str255);
void SetTextString (Str255 textStr);
```

Use the specified string as the text pane's text. This method uses `SetTextPtr` to set the text.

SetTextHandle

```
procedure SetTextHandle (textHand: Handle);
void SetTextHandle (Handle textHand);
```

Use the text `textHand` as the text for this text pane. This method uses `SetTextPtr` to set the text.

SetTextPtr

```
procedure SetTextPtr (textPtr: Ptr;
   numChars: longint);
void SetTextPtr (Ptr textPtr, long numChars);
```

Use the first `numChars` characters that `textPtr` points to as the text for this abstract text object. The default method does nothing. Your subclass must override this method so it makes a copy of the text.

GetTextHandle

```
function GetTextHandle: CharsHandle;
CharsHandle GetTextHandle (void);
```

Return a handle to the text of the abstract text. The default method does nothing. Your subclass must override this method. In most cases, the text

class should keep its text in a handle, and this method should return that handle—not a copy of it.

If your implementation requires that this method return a copy of the text in a handle, your subclass should override any methods that call `GetTextHandle` to dispose of the handle. In CAbstractText, the methods `GetCharBefore` and `GetCharAfter` expect `GetTextHandle` to return a handle to the actual text.

CopyTextRange

```
function CopyTextRange (start, end: long): Handle;
Handle CopyTextRange (long start, long end);
```

Return a copy of the text specified by `start` and end. The default method does nothing. Your subclass must override this method.

InsertTextPtr

```
procedure InsertTextPtr (text: Ptr; length: longint;
   fRedraw: Boolean);
void InsertTextPtr (Ptr text, long length,
   Boolean fRedraw);
```

Insert a copy of the given text at the start of the selection. `Text` is a pointer to the text. The default method does nothing. Your subclass must override this method.

InsertTextHandle

```
procedure InsertTextHandle (text: Handle;
   fRedraw: Boolean);
void InsertTextHandle (Handle text, Boolean fRedraw);
```

Insert a copy of the given text at the start of the selection. `Text` is a handle to the text.

GetCharBefore

```
procedure GetCharBefore (var aPosition: longint;
   var charBuf: tCharBuf);
void GetCharBefore (long *aPosition,
   tCharBuf charBuf);
```

Return the character before `aPosition`. The character and its size are returned in `charBuf`, and `aPosition` is updated with the starting position of the character. If there is no character after `aPosition`, then this method sets the length of the character to 0 (zero) and does not update `aPosition`.

GetCharAfter

```
procedure GetCharAfter (var aPosition: longint;
   var charBuf: tCharBuf);
void GetCharAfter (long *aPosition,
   tCharBuf charBuf);
```

Return the character after `aPosition`. The character and its size are returned in `charBuf`, and `aPosition` is updated with the starting position

of the character. If there is no character before `aPosition`, then this method sets the length of the character to 0 (zero) and does not update `aPosition`.

Text characteristics methods

SetFontNumber

```
procedure SetFontNumber (aFontNumber: integer);
void SetFontNumber (short aFontNumber);
```

Set the font by font number. The default method does nothing. Your subclass must override this method. If your subclass supports multiple fonts in a pane, this method should set the font for the current selection. Otherwise, it should set the font for the whole text pane.

SetFontName

```
procedure SetFontName (aFontName: Str255);
void SetFontName (Str255 aFontName);
```

Set the font by font name.

SetFontStyle

```
procedure SetFontStyle (aStyle: Style);
void SetFontStyle (short aStyle);
```

Set the font style. `AStyle` may be one of: `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, or `extend` The default method does nothing. Your subclass must override this method. If your subclass supports multiple styles in a pane, this method should set the style for the current selection. Otherwise, it should set the style for the whole text pane.

SetFontSize

```
procedure SetFontSize (aSize: integer);
void SetFontSize (short aSize);
```

Set the font size to the specified size. The default method does nothing. Your subclass must override this method. If your subclass supports multiple styles in a pane, this method should set the size for the current selection. Otherwise, it should set the size for the whole text pane.

SetTextMode

```
procedure SetTextMode (aMode: integer);
void SetTextMode (short aMode);
```

Set the text mode to the specified mode. `AMode` can be one of `srcOr`, `srcXor`, or `srcBic`. The default method does nothing. Your subclass must override this method.

SetAlignCmd

```
procedure SetAlignCmd (anAlignment: long);
void SetAlignCmd (long anAlignment);
```

Set the text alignment. `AnAlignment` can be one of `cmdAlignLeft`, `cmdAlignRight`, or `cmdAlignCenter`. The default method does noth-

ing. Your subclass must override this method. If your subclass supports multiple alignments in a pane, this method should set the alignment for the current selection. Otherwise, it should set the alignment for the whole text pane.

GetAlignCmd

```
function GetAlignCmd: longint;
long GetAlignCmd (void);
```

Return the current alignment. It can be one of `cmdAlignLeft`, `cmdAlignRight`, `cmdAlignCenter`, or `cmdNull`. The default method does nothing. Your subclass must override this method. If your subclass allows different alignments within a text pane and the current selection contains more than one alignment, this method should return `cmdNull`.

SetSpacingCmd

```
procedure SetSpacingCmd (aSpacingCmd: longint);
void SetSpacingCmd (long aSpacingCmd);
```

Set the space between lines of text. `ASpacingCmd` can be one of `cmdSingleSpace`, `cmd1HalfSpace`, or `cmdDoubleSpace`. The default method does nothing. Your subclass must override this method.

GetSpacingCmd

```
function GetSpacingCmd: longint;
long GetSpacingCmd (void);
```

Return the space between lines of text. It can be one of `cmdSingleSpace`, `cmd1HalfSpace`, or `cmdDoubleSpace`. The default method does nothing. Your subclass must override this method.

GetHeight

```
function GetHeight (startLine endLine: longint):
   longint;
long GetHeight (long startLine, long endLine);
```

Return the total height in pixels of the indicated lines of text. The default method does nothing. Your subclass must override this method.

Get1Height

```
function Get1Height (aLineNum: longint): integer;
short Get1Height (long aLineNum);
```

Return the height in pixels of the indicated line of text.

GetCharOffset

```
function GetCharOffset (aPt: LongPt): longint;
long GetCharOffset (LongPt *aPt);
```

Return the character position nearest the coordinate `aPt`. `APt` must be in frame coordinates. The default method does nothing. Your subclass must override this method.

GetCharPoint

```
procedure GetCharPoint (offset: long;
   var aPt: LongPt);
void GetCharPoint( long offset, LongPt *aPt);
```

Return the coordinates of the character position `offset`. `APt` is in frame coordinates. The default method does nothing. Your subclass must override this method.

GetCharStyle

```
procedure GetCharStyle (charOffset: long;
   var theStyle: TextStyle);
void GetCharStyle (long charOffset,
   TextStyle *theStyle);
```

Return style information for the character at position `charOffset`.

GetTextStyle

```
procedure GetTextStyle (var whichAttributes: integer;
   var aStyle: TextStyle);
void GetTextStyle (short *whichAttributes,
   TextStyle *aStyle);
```

Return current style information. `WhichAttributes` is a flag that indicates which text attributes to report on. The default method does nothing. Your subclass must override this method. If your subclass supports multiple styles in a text pane, this method should report only on the attributes that are continuous over the current selection, and set `whichAttributes` to those attributes.

The attributes flags and `TextStyle` record are described in *Inside Macintosh VI*, Chapter 15, "TextEdit."

Calibrating methods

SetWholeLines

```
procedure SetWholeLines (aWholeLines: Boolean);
void SetWholeLines (Boolean aWholeLines);
```

If `aWholeLines` is TRUE, this method will resize the frame so it will display only whole lines. The text pane is not redrawn.

GetWholeLines

```
function GetWholeLines: Boolean;
Boolean GetWholeLines (void);
```

Return the value of the `wholeLines` instance variable. If TRUE, the frame is set so it displays only whole lines and not partial lines.

ResizeFrame

```
procedure ResizeFrame (delta: Rect);
void ResizeFrame (Rect *delta);
```

Resize the frame when the size of its pane changes. The `delta` rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left.

FindLine

```
function FindLine (charPos: integer): longint;
long FindLine (short charPos);
```

Return the line number containing the specified character position. The default method does nothing. Your subclass must override this method.

Both line and character numbering start at zero. If the character position is before the start of the text (a negative number), this method should return zero. If the character position is beyond the end of the text, this method should return the number of the last line.

GetLength

```
function GetLength: longint;
long GetLength (void);
```

Return the length in bytes of the text buffer. The default method does nothing. Your subclass must override this method.

GetNumLines

```
function GetNumLines: longint
long GetNumLines (void);
```

Return the total number of lines in the text buffer. The default method does nothing. Your subclass must override this method.

AdjustCursor

Cursor method

```
procedure AdjustCursor (where: Point;
   mouseRgn: RgnHandle);
void AdjustCursor (Point where, RgnHandle mouseRgn);
```

Turn the cursor into an I-beam when the mouse is in the edit text pane.

CAppleEvent 10

Introduction

CAppleEvent is a class that contains an AppleEvent and its default reply. The THINK Class Library handles the four required AppleEvents (Open Application, Open Documents, Print Documents, and Quit Application) and lets you use others.

Heritage

Superclass	CObject
Subclasses	None

Using CAppleEvent

When your application receives an AppleEvent, the THINK Class Library puts in a CAppleEvent object and passes it to the gopher. You don't need to write any code to handle the four required AppleEvents. The application handles them by default.

If your application uses an AppleEvent besides the four required ones, you need to write a `DoAppleEvent` method to handle it. That method should be in the class that handles the event: your subclass of CPane, CDocument or CApplication. You don't need to subclass CAppleEvent to handle new AppleEvents.

To extract the parameters from your AppleEvent, you can send it either a `GetDescList` message, on page 160, which returns items in a `AEDescList`, or an `ExtractFromDescList` message, on page 160, which returns its items in a CArray object. After you extract all the parameters you know about, send your AppleEvent a `GotRequiredParam` message, on page 160, which returns TRUE if you extracted all the required parameters.

If your AppleEvent needs to interact with the user (e.g., display a dialog), send it the `RequestInteraction` message, on page 160. The THINK

Class Library assumes that the AppleEvents Open Application and Open Documents do not require user interaction. If they do, you must override CApplication's `DoAppleEvent` method

When your AppleEvent needs to return a result, send it a `SetErrorResult` message, described on page 161. If you need to store an error string or return data, send your AppleEvent object a `GetAEReply` message, described on page 160, and store that information directly in the reply.

How the THINK Class Library handles AppleEvents

When your initialize your application, it checks to see if the AppleEvent manager is installed. If it is, the switchboard installs a handler in the AppleEvents dispatch table, called `AppleEventHandler`, that handles all AppleEvents.

When your application is running and the switchboard receives a high-level event, it assumes the event is an AppleEvent and calls `AEProcessAppleEvent`. That function, in turn, calls the handler `AppleEventHandler`, which the switchboard installed earlier. The handler sends the switchboard a `DoAppleEvent` message. That method packages the AppleEvent and sends it to the gopher, which finds who will handle it.

Variables

These are internal instance variables which only subclasses of CAppleEvent should use. In THINK C, they are protected.

Variable	Type	Description
`theEvent`	`AppleEvent`	The AppleEvent.
`theReply`	`AppleEvent`	The default reply.
`theRefCon`	`longint`	The reference value.
`eventClass`	`DescType`	The event class; e.g., `'aevt'`.
`eventID`	`DescType`	The event ID; e.g., `'oapp'`.
`canInteract`	`Boolean`	TRUE if interaction with the user has previously been requested and approved.
`errCode`	`OSErr`	The error code for this event.

Variable	Type	Description
`notificationRec`	`NMRec`	The Notification Manager record for `AEInteractWithUser`. By default, it's NULL.
`idleProc`	`IdleProcPtr`	The idle procedure for `AEInteractWithUser`. If NULL, uses CSwitchboard's idle procedure

Methods

Creation and destruction

IAppleEvent

```
procedure IAppleEvent (theEvent, theReply: AppleEvent,
   theRefCon: longint, eventClass, eventID: DescType);
void IAppleEvent (const AppleEvent *theEvent,
   const AppleEvent *theReply, long theRefCon,
   DescType eventClass, DescType eventID);
```

Initialize a CAppleEvent object. `TheEvent` is the AppleEvent that CSwitchBoard received. `TheReply` is the default reply that the AppleEvent manager generates for an AppleEvent. `TheRefCon` is the reference value for this event's class and ID. `EventClass` and `eventID` are the event class and ID from `theEvent`.

Accessing methods

GetEventClass

```
function GetEventClass: DescType;
DescType GetEventClass (void);
```

Return the event class, e.g. `'aevt'`. The event class and event ID uniquely identify an AppleEvent.

GetEventID

```
function GetEventID: DescType;
DescType GetEventID (void);
```

Return the event ID, e.g. `'oapp'`. The event class and event ID uniquely identify an AppleEvent

GetAEEvent

```
function GetAEEvent: AppleEvent;
AppleEvent GetAEEvent (void);
```

Return the AppleEvent.

GetAEReply

```
function GetAEReply: AppleEvent;
AppleEvent GetAEReply (void);
```

Return the AppleEvent reply.

GetAERefCon

```
function GetAERefCon: longint;
long GetAERefCon (void);
```

Return the AppleEvent reference value for this event's class and ID. By default, this is always zero in the THINK Class Library. To use other values, override the `ISwitchboard` method in CSwitchboard, and change its call to `AEInstallEventHandler`.

GetDescList

```
procedure GetDescList (whichParam: AEKeyword;
   var descList: AEDescList);
void GetDescList (AEKeyword whichParam,
   AEDescList *descList);
```

Return the `AEDescList` associated with the keyword `whichParam`.

ExtractFromDescList

```
function ExtractFromDescList (whichParam: AEKeyword;
   itemType: DescType; itemSize: Size): CArray;
CArray *ExtractFromDescList (AEKeyword whichParam,
   DescType itemType, Size itemSize);
```

Get the descriptor list associated with the parameter `whichParam`, and puts its items into a CArray. This method is especially useful if all the items in the list are the same size and the data in the list is not keyword separated. This method does not use the keywords.

GotRequiredParams

```
function GotRequiredParams: Boolean;
Boolean GotRequiredParams (void);
```

Does the recommended check to determine if all the required parameters of an AppleEvent have already been extracted. You should call this method after you extract all the parameters you know about and before you actually handle the event.

RequestInteraction

```
function RequestInteraction (timeOutTicks: longint):
   OSErr;
OSErr RequestInteraction (long timeOutTicks);
```

Indicates that you need to interact with the user (e.g. put up a dialog) to handle the event. If interaction is allowed and the application is brought forward before the timeout period has elapsed, then this method returns `noErr` and you can proceed with the interaction. If any other value is returned, then your attempt to interact failed.

This method calls `AEInteractWithUser`, with `timeOutTicks`, an idle procedure, and a pointer to a Notification Manager record. By default, the idle procedure sends the switchboard an `AppleEventIdle` message, described on page 434. To use a different idle procedure, assign it to the `idleproc` instance variable. Also, the Notification Manager record pointer is NIL by default, so `AEInteractWithUser` supplies its own default record. To use your own record, assign it to the `notificationRec` instance variable.

SetErrorResult

```
procedure SetErrorResult (anErrorCode: OSErr);
void SetErrorResult (OSErr anErrorCode);
```

Set the result code for this event. If you successfully handle this function, set the result code to `noErr`. If you try to handle this function and fail, set the result code to appropriate error code.

GetErrorResult

```
function GetErrorResult: OSErr;
OSErr GetErrorResult (void);
```

Return the error code for this event. If the event was handled successfully, it will be `noErr`. If the event was not handled, it will be `errAEEventNotHandled`. If someone attempted to handle it, failed, and stored a result code, it will be some other result code. CSwitchboard uses a special exception handler to trap exceptions raised during the handling of an AppleEvent, and will return the exception error instead.

CApplication 11

Introduction

Every program must create a subclass of CApplication and create only one instance of this class. The application is the highest level in the chain of command. It is the only bureaucrat without a supervisor.

Heritage

Superclass	CDirectorOwner
Subclasses	You must define a subclass of this class.

Using CApplication

CApplication is one of the classes you must override to implement an application with the THINK Class Library. An application usually has one or more subordinate **directors** which handle the interaction between commands and windows. The most common kind of director is a **document** which, in addition to a window, also handles a file.

The application and the chain of command

The application is the root of the chain of command. If the current command object, or **bureaucrat**, (stored in the global variable `gGopher`) can't handle a command, it passes the command to its supervisor. If no bureaucrat can handle the message, it ends up with the application. If the application doesn't handle a command, the message is ignored.

Writing the main program

Your main program must create an application object and assign it to the global variable `gApplication`. After initializing your application, send it application a `Run` message to get it started. Finally, send your application an

`Exit` (or `ExitApp` in Pascal) message to give it a chance to clean up after itself. In THINK Pascal, your main routine should look like this:

In this example, `CYourApp` is the name of your application subclass.

```
begin
   new(CYourApp(gApplication));
   CYourApp(gApplication).IYourApp;
   gApplication.Run;
   gApplication.ExitApp;
end.
```

And in THINK C, your main routine should look like this:

```
void main()
{
   gApplication = new(CYourApp);
   ((CYourApp *)gApplication)->IYourApp();
   gApplication->Run();
   gApplication->Exit();
}
```

Your application subclass must override or define these five methods:

initialization method (`IYourApp` in the example)
`OpenDocument`
`SetUpFileParameters`
`DoCommand`
`CreateDocument`

Of course, your subclass can override additional methods if it needs to.

Handling low memory situations

The term "rainy day fund" comes from the expression "saving for a rainy day." It means to save money in case of disaster.

The CApplication class provides several methods that deal with low memory situations. These methods use a memory reserve called the **rainy day fund**.

When you call `IApplication`, you specify how much memory your application should allocate for the rainy day fund. You also specify how many bytes of that fund should be available to satisfy Toolbox routine memory requests and how many bytes of that fund should be available to satisfy critical operation requests. These values are stored in the instance variables `toolboxBalance` and `criticalBalance`.

If the Macintosh Memory Manager gets a request for more memory than is available, it calls a grow zone function. In the THINK Class Library, the grow zone function sends the application a `GrowMemory` message.

The `GrowMemory` method tries several strategies to free memory in the heap. First, it sends the application a `MemoryShortage` message. Your application subclass should override this method to release memory that is not crucial to execution. For instance, your `MemoryShortage` method might release a code segment it's not using, or it might dispose of a buffer it no longer needs. If the `MemoryShortage` method wasn't able to release enough memory, `GrowMemory` starts using the rainy day fund.

`GrowMemory` looks first at the `inCriticalOperation` instance variable to relase memory from the rainy day fund:

If inCriticalOperation is	leave this much in reserve
TRUE	`toolboxBalance`
FALSE	`criticalBalance`

A critical operation is a temporary operation that takes place in a single Macintosh event, that might take a lot of memory, and that should not cause the application to crash. For instance, saving a file is a critical operation. You can use the routine `SetCriticalOperation` to set the `inCriticalOperation` flag.

If `GrowMemory` still wasn't able to release enough memory and the canFail flag is TRUE, `GrowMemory` returns without trying to allocate any more memory. If the `canFail` flag is TRUE, the application is saying that it can deal with a failing memory request.

If the `canFail` flag is FALSE, GrowMemory tries to release all the memory left in the rainy day fund because the application is not prepared to deal with a failing memory request. If there is not enough memory, even after using the rainy day fund, it is likely that the application will crash.

You can use the routine `SetAllocation` to set and reset the `canFail` flag. In general, your application should be prepared to handle failing memory requests.

Variables

The global variable `gApplication` points to your application object. The method `IApplication` sets this variable.

The instance variables of the application class handle events, memory shortage situations, standard file parameters, and flow of control. Your application subclass may define additional instance variables.

Global variable

Variable	Type	Description
gApplication	CApplication	The single instance of your application object.
gSystem	tSystem	A global record with information about the Macintosh and System that your program is running with.

Only methods of CApplication and its subclasses should use this variable. In THINK C, it's a class variable.

Variable	Type	Description
cMaxSleepTime	long	default for WaitNextEvent

Event related instance variables

The event related instance variables handle the interaction between the Macintosh Toolbox Event Manager and your application. Your application subclass should not manipulate these variables.

Variable	Type	Description
itsSwitchboard	CSwitchboard	Points to the event-processing object.
itsIdleChores	CList	Chores to perform at idle time.
itsUrgentChores	CCluster	Chores to perform after the current event.
urgentsToDo	Boolean	Are any urgent chores pending?
running	Boolean	If TRUE, the program is still running.
unhandledTask	CTask	Task that no document handled

Phase related instance variables

This variable tells you what phase your application is in.

Variable	Type	Description
phase	integer	The phase the application is in.

These are its possible values for phase:

Value	Description
appInitializing	The application is initializing.
appRuning	The application is running.
appQuitting	The application is quitting.

Memory related instance variables

The memory related variables are used when your application is running out of memory. The rainyDayFund variable specifies the number of bytes to set aside to use when your application runs into a critical memory situation. You specify this amount in your application initialization method.

Variable	Type	Description
rainyDayFund	Size	Bytes of memory to set aside for critical memory situations.
criticalBalance	Size	Portion of rainyDayFund to use for critical operations.
toolboxBalance	Size	Portion of rainyDayFund to use for Toolbox operations that must not fail.
tempAllocation	Size	Portion of rainyDayFund that a routine has borrowed.
rainyDay	Handle	Handle to the reserve memory.
rainyDayUsed	Boolean	Has rainy day fund been used?
memWarningIssued	Boolean	Has the user been alerted?
canFail	Boolean	Is it OK for memory request to fail?
inCriticalOperation	Boolean	Is it OK to use up to criticalBalance bytes from reserve?

Standard file instance variables

The standard file variables are used in the `ChooseFile` method as parameters to the Macintosh Toolbox `SFPGetFile` routine. You set these variables in your `SetUpFileParameters` method.

Variable	Type	Description
`sfNumTypes`	`integer`	Number of file types recognized
`sfFileTypes`	`SFTypeList`	File types that the application regognizes
`sfFileFilter`	`FileFilterProcPtr`	Filter for files to display
`sfGetDLOGHook`	`DlgHookProcPtr`	Hook for handling get dialog
`sfGetDLOGid`	`integer`	Dialog resource ID for get file. Default is `getDlgID` (-4000).
`sfGetDLOGFilter`	`ModalFilterProcPtr`	Filter for get dialog events

Methods

The methods of the CApplication class deal with application initialization and document handling. Since the application is the ultimate supervisor for all the command objects, it also serves as the end of the chain of command.

Initialization methods

The initialization methods set up the application's memory and file parameters. Your application subclass should implement its own initialization method and call the default initialization method.

IApplication

```
procedure IApplication (extraMasters: integer;
   aRainyDayFund, aCriticalBalance,
   aToolboxBalance: Size);
void IApplication (short extraMasters,
   Size aRainyDayFund, Size aCriticalBalance,
   Size aToolboxBalance);
```

This is the main initialization method. In your application subclass, you should implement a method called `I`*MyAppClass*, where *MyAppClass* is the name of your application subclass.

The `extraMasters`, `aCriticalBalance`, and `aToolboxBalance` parameters are passed to the `InitMemory` method described later on.

Your application subclass must call `CApplication.IApplication` (in THINK Pascal) or `CApplication::IApplication` (in THINK C) in its initialization method. If your application subclass defines instance variables, your initialization method should initialize the variables.

This method also initializes these global variables. All the global variables are described in detail in Chapter 72, "Global Variables."

`gApplication`	`gSignature`
`gSleepTime`	`gIBeamCursor`
`gWatchCursor`	`gUtilRgn`
`gGopher`	`gLastViewHit`
`gClicks`	`gSystem`

Finally, this method sends the following messages to your application object. The rest of this section goes into detail for each method.

`InitToolbox`	`InitMemory`
`InspectSystem`	`InstallPatches`
`MakeSwitchboard`	`MakeError`
`MakeDesktop`	`MakeClipboard`
`MakeDecorator`	`SetUpFileParameters`
`SetUpMenus`	

InitToolbox

```
procedure InitToolbox;
void InitToolbox (void);
```

This method initializes all of the Macintosh Toolbox managers. Your application subclass should not need to override this method.

InitMemory

```
procedure InitMemory (extraMasters: integer;
   aRainyDayFund, aCriticalBalance,
   aToolboxBalance: Size);
void InitMemory (short extraMasters,
   Size aRainyDayFund, Size aCriticalBalance,
   Size aToolboxBalance);
```

This method initializes the Macintosh memory manager and sets up the `GrowZone` function used in low memory situations. `ExtraMasters` is the number of times to call the Toolbox routine `MoreMasters`. `ARainyDayFund` is the number of bytes to set aside in the application heap to deal with low memory situations. `ACriticalBalance` is the number of bytes to use from the `rainyDayFund` for critical operations. `AToolboxBalance` is the

number of bytes to use from the `rainyDayFund` for Toolbox operations that can't fail.

`ARainyDayFund` must be greater than or equal to `aCriticalBalance`, and `aCriticalBalance` must be greater than `aToolboxBalance`.

The values that you use as arguments to this method depend on the memory requirements of your application. You can use these values to get started:

Argument	Suggested value
`extraMasters`	10
`aRainyDayFund`	45000
`aCriticalBalance`	40000
`aToolboxBalance`	20000

MakeSwitchboard

```
procedure MakeSwitchboard;
void MakeSwitchboard (void);
```

This method creates the application's switchboard and stores it in the instance variable `itsSwitchboard`. If you create your own subclass of CBartender, you should override this method in your application subclass.

MakeError

```
procedure MakeError;
void MakeError (void);
```

This method creates the error-handler and stores it in the global variable `gError`. If you create your own error-handler, you can create a subclass of CError and override this method to use your error-handler.

MakeDesktop

```
procedure MakeDesktop;
void MakeDesktop (void);
```

This method creates the desktop and stores it in the global variable `gDesktop`. The default method creates the standard desktop CDesktop. If your application uses a non-standard desktop, such as CFWDesktop, you should override this method in your application subclass.

CFWDesktop, which implements a desktop with floating windows, is described on page 315.

MakeClipboard

```
procedure MakeClipboard;
void MakeClipboard (void);
```

This method creates the clipboard and stores it in the global variable `gClipboard`. If you create your own subclass of CClipboard, you should override this method to create a clipboard of your class.

MakeDecorator

```
procedure MakeDecorator;
void MakeDecorator (void);
```

This method creates the window decorator and stores it in the global variable `gDecorator`. The window decorator is responsible for the default sizes of windows and arranges them neatly on the desktop. If you want to implement a different decorator, you can create a subclass of CDecorator and override this method to use your decorator.

SetUpFileParameters

```
procedure SetUpFileParameters;
void SetUpFileParameters (void);
```

This method lets you set up the parameters that specify what kinds of files your application works on. The THINK Class Library passes some of these parameters to the Toolbox `SFPGetFile` function which displays the standard get file dialog. Your application subclass should override this method and call the inherited method to set up the default values.

Your own method should set the following instance variables and globals:

`sfNumTypes`	The number of different types of files your application deals with. If your application can open any kind of file, use –1.
`sfFileTypes`	The file types of the files that your application deals with. One type for each element of this array.
`gSignature`	The four-character signature of your application. Although it's not an instance variable, this method is the place to set up this global variable.
`sfFileFilter`	Optional. A pointer to a function that filters the file names your application can deal with.
`sfGetDLOGHook`	Optional. A pointer to a dialog hook function for the standard get file dialog.
`sfGetDLOGid`	Optional. The resource ID of the standard get file dialog you're using. Do not change this variable if you want to use the default dialog
`sfGetDLOGFilter`	Optional. A pointer to a dialog filter proc

To learn about file filter functions, dialog hook functions, and dialog filters, see *Inside Macintosh I*, Chapter 20, "The Standard File Package."

SetUpMenus

```
procedure SetUpMenus;
void SetUpMenus (void);
```

SetUpMenus calls MakeBartender and reads all the menu information from your application's MBAR 1 resource and builds the menus. It also builds the desk accessory menu.

If your application uses menus that need to be built on the fly, you should override this method. For instance, if your application uses a **Font** menu, you would build it in this method. Be sure to call inherited SetUpMenus to make sure all the regular menus get built. See the description of CBartender, especially section "Resource based menus" on page 194, for more information about building these kinds of menus.

MakeBartender

```
procedure MakeBartender;
void MakeBartender (void);
```

This method creates the bartender object, which handles all interactions with the menu bar and menu items, and stores it in the global variable gBartender. If you create your own subclass of CBartender, you should override this method in your application subclass.

InspectSystem

```
procedure InspectSystem;
void InspectSystem (void);
```

This method fills in the fields of the gSystem global variable which gives you information about the Macintosh and the System Software that your application is running under.

In THINK C, tSystem is declared like this:

```
typedef struct
{
   Boolean  hasWNE          : 1;
   Boolean  hasColorQD      : 1;
   Boolean  hasGestalt      : 1;
   Boolean  hasAppleEvents  : 1;
   Boolean  hasAliasMgr     : 1;
   Boolean  hasEditionMgr   : 1;
   Boolean  hasHelpMgr      : 1;
   Boolean  hasScriptMgr    : 1;
   Boolean  hasFPU          : 1;
   short    scriptsInstalled;
   short    systemVersion;
} tSystem;
```


In THINK Pascal it's defined like this:

```
type
   tSystem = packed record of
      hasWNE,
      hasColorQD,
      hasGestalt,
      hasAppleEvents,
      hasAliasMgr,
      hasEditionMgr,
      hasHelpMgr,
      hasScriptMgr,
      hasFPU              : Boolean;
      scriptsInstalled,
      systemVersion       : integer;
   end;
```

The field scriptsInstalled tells you how many scripts are in use. The field systemVersion gives you the version of the Macintosh System that's running. The version number is given as two byte-long numbers. For example, if systemVersion is 0x0607, the System version number is 6.0.7.

Advanced initialization methods

You should not need to override these methods. If you are an advanced Macintosh programmer, you may want to take advantage of the fact that they're called in IApplication.

InstallPatches

```
procedure InstallPatches;
void InstallPatches (void);
```

Install patches to Toolbox routines. The default method patches LoadSeg to make sure that CODE resources can be loaded. If a code resource can't be loaded, the patch raises an exception. The default method also patches ExitToShell to call RemovePatches.

InstallPatches

```
procedure RemovePatches;
void RemovePatches (void);
```

Remove the Toolbox patches made in InstallPatches.

ForceClassReferences

```
procedure ForceClassReferences;
void ForceClassReferences (void);
```

If there are classes in your application that are instantiated by name, add a dummy reference to them here to prevent the smart linker from stripping them out. .

Accessing methods

GetPhase

```
function GetPhase: integer;
short GetPhase (void);
```

Return current application phase. The possible values are `appInitializing`, `appRunning`, and `appQuiting`.

Command methods

The command methods handle events that the application takes care of. Since messages frequently get passed up the chain of command— from the pane, to the document, to the application— the application is the last chance to handle command messages. Most of the command methods don't do anything. They're null methods that keep the message from being passed on.

Notify

```
procedure Notify (theTask: CTask);
void Notify (CTask *theTask);
```

When a subordinate object finishes a task, it sends the task to its supervisor. Documents usually handle `Notify` messages. This method is here in case a document tries to pass the `Notify` message on to the application. It places `theTask` into the instance variable `unhandledTask`. The task is disposed of at the end of the current event.

DoKeyDown

```
procedure DoKeyDown (theChar: char; keyCode: Byte;
   macEvent: EventRecord);
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

You should handle key-down events within a pane or a document. For applications, this method doesn't do anything. It's here in case the `DoKeyDown` message gets passed up the chain of command. Your application subclass should not override this method.

DoAutoKey

```
procedure DoAutoKey (theChar: char; keyCode: Byte;
   macEvent: EventRecord);
void DoAutoKey (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

You should handle auto-key events within a pane or a document. For applications, this method doesn't do anything. It's here in case the `DoAutoKey` message gets passed up the chain of command. Your application subclass should not override this method.

DoKeyUp

```
procedure DoKeyUp (theChar: char; keyCode: Byte;
   macEvent: EventRecord);
void DoKeyUp (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

You should handle key-up events within a pane or a document. For applications, this method doesn't do anything. It's here in case the `DoKeyUp` message gets passed up the chain of command. Your application subclass should not override this method.

Note

The Macintosh system event mask is usually set to mask out key-up events. If you need to get key-up events, be sure to reset the system event mask with the Toolbox routine `SetEventMask`.

DoCommand

```
procedure DoCommand (theCommand: longint);
void DoCommand (long theCommand);
```

This is the only application command method that really does anything. `DoCommand` handles application commands that the user chooses from the menu. These are the commands that the default `DoCommand` method handles:

Command	Action
`cmdNew`	Sends a `CreateDocument` message to the application.
`cmdOpen`	Sends a `ChooseFile` message to your application. If the reply is good, sends an `OpenDocument` message to your application. The default `ChooseFile` message calls `SFPGetFile` with the values you specified in the `SetUpFileParameters` method. Your application must override the `OpenDocument` method.
`cmdClose`	If the front window is a desk accessory, close it. Your document class should handle this command to close documents.
`cmdQuit`	Sends a `Quit` message to your application. The default `Quit` method sends a

	`Quit` message to the supervisor (a director) of each open window
`cmdUndo`	
`cmdCut`	
`cmdCopy`	
`cmdPaste`	
`cmdClear`	If the front window is a desk accessory, always call `SystemEdit`. Your document class should handle these commands to edit documents.
`cmdToggleClip`	Sends a `Toggle` message to the clipboard.

If your application defines its own application-related events, your application class should override this method. If your `DoCommand` method gets a command that it does not handle, it should call `inherited DoCommand`.

UpdateMenus

```
procedure UpdateMenus;
void UpdateMenus (void);
```

This method enables the appropriate menu items right before a menu selection. The default method enables the **Quit** command. If the application is in the foreground, it enables the **Show/Hide Clipboard** command. If the application is in the background, this method enables the `cmdClose`, `cmdUndo`, `cmdCut`, `cmdCopy`, `cmdPaste`, and `cmdClear` commands for desk accessories.

Your document's `UpdateMenus` method should enable the appropriate **Edit** menu commands for the document.

PackageAppleEvent

```
function PackageAppleEvent (
   theEvent, theReply: AppleEvent; long theRefCon;
   eventClass, eventID: DescType): CAppleEvent;
CAppleEvent *PackageAppleEvent (
   struct AppleEvent *theEvent,
   struct AppleEvent *theReply, long theRefCon,
   DescType eventClass, DescType eventID);
```

Package an incoming AppleEvent and its default reply into a CAppleEvent object. This method returns a CAppleEvent object of class. If you create a subclass of CAppleEvent, you need to override this class. CAppleEvent is described on page 157.

DoAppleEvent

```
procedure DoAppleEvent (anAppleEvent: CAppleEvent);
void DoAppleEvent (CAppleEvent *anAppleEvent);
```

Respond to an AppleEvent. This method handles the four required AppleEvents: `kAEOpenApplication`, `kAEOpenDocuments`, `kAEPrintDocuments`, and `kAEQuitApplication`. If you create a subclass of CAppleEvent, you need to override this class. CAppleEvent is described on page 157.

Memory management methods

These methods deal with critical memory situations in your application. The `InitMemory` method sets the function `GrowZoneFunc` as the function to call in low-memory situations. This function sends a `GrowMemory` message to your application to try to reclaim enough memory to continue.

Note

`GrowZoneFunc` is defined in `TCL.p` in THINK Pascal and in `CError.h` in THINK C.

RequestMemory

```
procedure RequestMemory (aCanFail: Boolean);
void RequestMemory (Boolean aCanFail);
```

Change the `canFail` instance variable which controls how `GrowMemory` releases memory from the rainy day fund. If `aCanFail` is FALSE, the `GrowMemory` method is more conservative about releasing memory for a failing memory request. If `aCanFail` is TRUE, `GrowMemory` will exhaust the rainy day fund to satisfy the request.

Instead of using this method, you should use the utility routine `SetAllocation` (described on page 492). `SetAllocation` sends a `RequestMemory` message to the application and returns the previous value, so you can set it back.

Here's an example, in THINK Pascal:

```
procedure AClass.AMethod;
   var
      oldAlloc: Boolean;
      myHandle: Handle;
begin
   oldAlloc := SetAllocation(TRUE);
   myHandle := NewHandle(50000);
   SetAllocation(oldAlloc);

   if myHandle := nil then
      begin
         couldn't get 50000 bytes
      end
   else
      begin
         Go on with this operation
      end
end;
```

And here's the same example in THINK C:

```
void AClass::AMethod(void);
{
   Boolean oldAlloc;

   oldAlloc = SetAllocation(TRUE);
   myHandle = NewHandle(50000L);
   SetAllocation(oldAlloc);

   if (myHandle == NULL) {
      couldn't get 50000 bytes
   }
   else {
      Go on with this operation
   }
}
```

Remember that if the argument you pass to `SetAllocation` or `RequestMemory` is FALSE, `GrowMemory` will do whatever it can to get the memory. If there still isn't enough memory after the rainy day fund has been exhausted, your program will probably crash.

SetCriticalOperation

```
procedure SetCriticalOperation (isCritical: Boolean);
void SetCriticalOperation (Boolean isCritical);
```

Set the `inCriticalOperation` instance variable which controls how `GrowMemory` releases memory from the rainy day fund. If inCriticalOperation is TRUE, and there is a memory shortage that causes GrowMemory to be called, grow memory tries to release as much memory in the rainy day fund that would leave toolboxBalance bytes free.

Your application should call the utility routine `SetCriticalOperation` (described on page 493) instead of sending a `SetCriticalOperation` message to `gApplication`.

Use this routine when there is a operation that must complete and for which your application might need to use some of the memory in the rainy day fund. The critical operation should release any memory that it allocated.

For example, when your application is saving a file, you may need to allocate an extraordinary amount of memory. In this case, you would call `SetCriticalOperation` to let `GrowMemory` know that, if necessary, you're willing to use more of the rainy day fund. Once the operation is complete, you would dispose of the memory.

Note that the `Run` method resets `inCriticalOperation` to FALSE every time through the event loop.

GrowMemory

```
function GrowMemory(bytesNeeded: Size): longint;
long GrowMemory (Size bytesNeeded);
```

The `GrowZoneFunc` function that the Memory Manager calls in low memory situations invokes this method to try to reclaim memory. `GrowMemory` sends a `MemoryShortage` message to the application.

If `MemoryShortage` wasn't able to release enough memory, `GrowMemory` starts to use the rainy day fund. If `inCriticalOperation` is TRUE, it uses as much of the rainy day fund that would still leave `toolboxBalance` bytes in the fund. If `inCriticalOperation` is FALSE, it uses as much that would leave `citicalBalance` bytes in the fund.

If `GrowMemory` still wasn't able to release enough memory from the rainy day fund, and `canFail` is FALSE, the entire rainy day fund is released. If the entire rainy day fund has been exhausted, this method calls `OutOfMemory`.

MemoryShortage

```
procedure MemoryShortage (bytesNeeded: Size);
void MemoryShortage (Size bytesNeeded);
```

The `GrowMemory` method sends a `MemoryShortage` message to your application to try to free memory. The default `MemoryShortage` method does nothing, so your application subclass should override this method. Your `MemoryShortage` method should try to free `bytesNeeded` bytes of memory, and it should disable menu commands that won't work in low memory situations.

Warning
Your MemoryShortage method must not allocate any memory.

MemoryReplenished

```
procedure MemoryReplenished;
void MemoryReplenished (void);
```

Your application will get this message when the memory situation is no longer critical. The default MemoryReplenished method does nothing, so your application subclass should override this method. Your MemoryReplenished method should enable the commands you disabled in the MemoryShortage method. You might also want to reallocate memory that you released in that method.

OutOfMemory

```
function OutOfMemory (bytesNeeded: Size): longint;
long OutOfMemory (Size bytesNeeded);
```

A memory request cannot be satisfied, and canFail was FALSE, so the application won't handle the memory allocation failure. This method calls Failure(memFullError, 0). If there is any memory left, the top-level handler will display an alert, but it is more likely that the application will crash.

You can override this method if you can free up more memory, or if you can exit the application gracefully without allocating memory. Normally, you would try to free enough memory in your MemoryShortage method. If there are more drastic ways to release memory, implement them here. This method should return 1 if it successfully released memory or zero if it couldn't.

Warning
This method should not allocate any memory or call any routines that allocate memory.

Execution methods

The execution methods are invoked while your application is running. Most of these methods handle system-related events. The only execution method your application should override is the Exit method.

Run

```
procedure Run;
void Run (void);
```

This method runs your application until the user quits. This method sends the application ProcessEvent messages until the user chooses to quit.

Before running your program, this method sends a Preload message to your application to open or print documents that the user selected and opened from the Finder.

Your application should not override this method.

ProcessEvent

```
procedure ProcessEvent;
void ProcessEvent (void);
```

Process and dispatch one event. This method sends a ProcessEvent message to the switchboard (stored in the instance variable itsSwitchboard). If you installed an urgent chore, this method sends it a Perform message.

Preload

```
procedure Preload;
void Preload (void);
```

If the user opened or chose to print files from the Finder, this method sends the application an OpenDocument message for each document. If the user chose the **Print** command, this method sends a DoCommand(cmdPrint) message to the gopher. After processing all the files, this method sends the application a StartUpAction message.

StartUpAction

```
procedure StartUpAction (numPreloads: integer);
void StartUpAction (short numPreloads);
```

This method gives you an opportunity to perform any startup actions. NumPreloads is the number of files that the user selected from the Finder. If numPreloads is zero, the default method sends a DoCommand(cmdNew) message to the gopher (usually the application). The effect of the default method is to open an untitled document at startup.

Suspend

```
procedure Suspend;
void Suspend (void);
```

Your application is about to be suspended under MultiFinder. The default method calls the inherited method and sets the global variable gInBackground to TRUE.

Resume

```
procedure Resume;
void Resume (void);
```

Your application has come back to the foreground under MultiFinder. The default method calls the inherited method and sets the global variable gInBackground to FALSE.

SwitchToDA

```
procedure SwitchToDA;
void SwitchToDA (void);
```

A desk accessory is becoming active. The default method sends the application a `Suspend` message. Your application should not override or use this method.

SwitchFromDA

```
procedure SwitchFromDA;
void SwitchFromDA (void);
```

Your application is becoming active after a desk accessory was active. The default method sends your application a `Resume` message. Your application should not override or use this method.

Idle

```
procedure Idle (macEvent: EventRecord);
void Idle (EventRecord *macEvent);
```

This method handles periodic tasks. It also checks to see if a critical memory situation is no longer critical. `Idle` sends `Dawdle` messages to the gopher and to each of the gopher's supervisors. It also sends `Perform` messages to all chores. Your application should not override this method.

Quit

```
procedure Quit;
void Quit (void);
```

The user wants to quit the application. This method sends a `Quit` message to the supervisor (a director) of each open window. Any of the directors can cancel quitting. Your application should not override this method.

Note

The `Quit` method for directors returns a Boolean value. If it returns `FALSE`, quitting is canceled.

Exit/ExitApp

```
procedure ExitApp;
void Exit (void);
```

The application is about to exit. Your main program should send an `Exit` message to the application before it terminates; the THINK Class Library will not send the message for you. The default method does nothing.

The `Exit` method is the last chance you have to clean up after your application. For example, this is a good place to delete any temporary files you've created.

JumpToEventLoop

```
procedure JumpToEventLoop;
void JumpToEventLoop (void);
```

This method is for compatibility with earlier versions of the THINK Class Library. It callse `Failure(kSilentErr, 0)` to force a return to the top-level exception handler in `Run`.

Document methods

The document methods of an application deal with creating and opening documents. In your application subclass, the only document methods you'll need to override are `CreateDocument` and `OpenDocument`. All the other document methods are used internally to implement standard behavior.

CreateDocument

```
procedure CreateDocument;
void CreateDocument (void);
```

This method creates a new document. The default `DoCommand` method sends a `CreateDocument` message to the application when the user chooses **New** from the **File** menu. Your application must override this method. Your `CreateDocument` method needs to create a new document, initialize it, and then send it a `NewFile` message.

OpenDocument

```
procedure OpenDocument (macSFReply: SFReply);
void OpenDocument (SFReply *macSFReply);
```

This method opens an existing document. The `macSFReply` record specifies which document to open. The default `DoCommand` method sends an `OpenDocument` message to the application when the user chooses **Open...** from the **File** menu.

Note

`DoCommand` actually sends a `ChooseFile` message to the application first, and if the reply is good, it sends an `OpenDocument` message. Your `OpenDocument` method can assume that the `macSFReply` record is valid.

Your application subclass must override this method. Your `OpenDocument` method needs to create a new document, initialize it, and then send it an `OpenFile(macSFReply)` message.

ChooseFile

```
procedure ChooseFile (var macSFReply: SFReply);
void ChooseFile (SFReply *macSFReply);
```

This method displays a standard get file dialog and places the reply in the `macSFReply` record. This method calls the Toolbox routine `SFPGetFile`

with the file parameters you specified in the `SetUpFileParameters` method.

You can send the application a `ChooseFile` message from other methods to get the name and location of a file. Don't forget to check the `macSFReply.good` field to make sure that the information in the record is valid.

Your application subclass should not override this method unless it uses a different technique to open a document.

Periodic task methods

AssignIdleChore

```
procedure AssignIdleChore (theChore: CChore);
void AssignIdleChore (CChore *theChore);
```

This method adds `theChore` to the application's list of chores. The application sends a `Perform` message to each chore at idle time. Your application should not override this method.

Note

If you want to remove the chore from the list later on, you must keep a pointer to the chore object.

CancelIdleChore

```
procedure CancelIdleChore (theChore: CChore);
void CancelIdleChore (CChore *theChore);
```

This method removes `theChore` from the application's list of chores. Your application should not override this method.

AssignUrgentChore

```
procedure AssignUrgentChore (theChore CChore);
void AssignUrgentChore (CChore *theChore);
```

This method adds `theChore` to the application's list of urgent chores. The application sends a `Perform` message to each urgent chore after handling the current event and then removes the chore. Your application should not override this method.

Class resources

Resource	Description
MBAR 1	The application's menu bar
STR# 129	Low memory warning strings.

CArray 12

Introduction

CArray implements a resizable array.

Heritage

Superclass	CCollection
Subclasses	CCluster
	CRunArray

Using CArray

Use CArray when you want an array you can resize. The elements may be any size, as long as they're all the same size.

To insert a new element into the array, use `InsertAtIndex`. It moves the element in that slot and all the element below it down one slot. It's illustrated in Figure 12-1.

Figure 12-1 Before and after InsertAtIndex

To update the value of an existing element, use `SetItem`. It at that slot with the new element. It's illustrated in Figure 12-2.

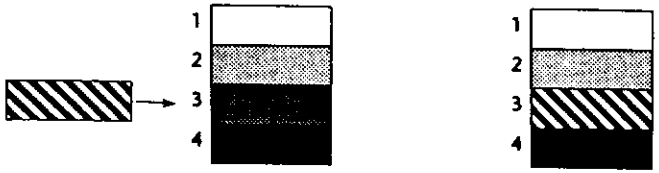

Figure 12-2 Before and after SetItem

To delete an element, use `DeleteItem`. It deletes the element and moves all the elements below it up one slot.

CArray stores the elements in its array in one large piece of memory that it allocates dynamically. It uses the instance variable `blockSize` to control how to grow and shrink this memory. If there are no empty slots when you try to insert an element, CArray grows memory by `blockSize` slots. When you delete an element and there are `blockSize` empty slots left, CArray shrinks the memory by `blockSize` slots. By default, `blockSize` is 3.

CArray has two methods that move elements in the array: `Swap` and `MoveItemToIndex`. To store an element while moving it, CArray has a temporary storage slot at the end of the array. To make sure an element in temporary storage isn't accidently overwritten, those methods set the flag `usingTemporary` whenever they put something in the temporary storage slot. This slot is not counted in the `slots` instance variable, which is the total number of slots that can permanently store elements.

Variables

Variable	Type	Description
`blockSize`	`integer`	Number of slots to allocate when more space is needed.
`slots`	`longint`	Total number of slots allocated.
`hItems`	`Handle`	Items in the array
`elementSize`	`longint`	Size of each element in bytes
`lockChanges`	`Boolean`	If `TRUE`, you can't insert or delete in array.
`usingTemporary`	`Boolean`	If `TRUE`, temporary element storage buffer is in use.

Methods

Creation and destruction methods

IArray

```
procedure IArray (elementSize: longint);
void IArray (long elementSize);
```

Initialize the array. `ElementSize` is the size of each element in the array. This method sets the size of each element in the array to `ElementSize`, sets blockSize to 3, and allocates enough space for the temporary storage slot.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the array and the block of memory it allocated.

Accessing methods

SetBlockSize

```
procedure SetBlockSize (aBlockSize: integer);
void SetBlockSize (short aBlockSize);
```

Set the number of elements to add to the array when CArray allocates more space. `IArray` initializes this value to 3.

SetLockChanges

```
function SetLockChanges (fLockChanges: Boolean):
   Boolean;
Boolean SetLockChanges (Boolean fLockChanges);
```

If `fLockChanges` is TRUE, you can't use the `InsertAtIndex` and `DeleteItem` methods. This method sets `lockChanges` to

fLockChanges and returns the previous value of lockChanges. CCluster uses this method to prevent you from corrupting the array when you iterate over its elements with DoForEarch.

Insertion and deletion methods

InsertAtIndex

```
procedure InsertAtIndex (itemPtr: Ptr;
   index: longint);
void InsertAtIndex (void *itemPtr, long index);
```

Insert the element at itemPtr at the index index. If index is already occupied, move the element in that slot and all the element below it down one slot. If index is beyond the last position, add the item to the end of the array. If necessary, this method allocates a larger block of memory for the array. This method sends a BroadcastChange message with arrayInsertElement as the reason.

DeleteItem

```
procedure DeleteItem (index: longint);
void DeleteItem (long index);
```

Delete the element at index index. This method moves all the elements below index up one slot. If the number of free slots is greater than blockSize, this method reallocates a block of memory that's smaller by blockSize slots.

SetItem

```
procedure SetItem (itemPtr: Ptr; index: longint);
void SetItem (void *itemPtr, long index);
```

Put the element at itemPtr into the array at position index. Index must be within the array. This method sends a BroadcastChange message with arrayElementChanged as the reason.

Store

```
procedure Store (itemPtr: Ptr; index: longint);
void Store (void *itemPtr, long index);
```

Put the element at itemPtr into the array at position index. Whenever possible, use Set instead. This is an internal method that performs no error-checking. In THINK C, this is a protected method.

Membership method

GetItem

```
procedure GetITem (itemPtr: Ptr; index: longint);
void GetItem (void *itemPtr, long index);
```

Return the element at position index in the block memory at itemPtr. Index must be within the array. ItemPtr must point to a block large enough to hold one element.

Retrieve

```
procedure Retrieve (itemPtr: Ptr; index: longint);
void Retrieve (void *itemPtr, long index);
```

Return the element at position index in the block memory at itemPtr. Whenever possible, use Get instead. This is an internal method that performs no error-checking. In THINK C, this is a protected method.

Search

```
function Search (itemPtr: Ptr; function compare (
   item1, item2: Ptr): integer): Ptr;
void *Search (void *itemPtr, CompareFunc compare);
```

Find an element that satisfies a condition. The function compare tests for the condition. It takes two arguments. The first is itemPtr and the second is a pointer to an element in the array. Compare should return 0 when an element satisfies the condition. ItemPtr is a pointer to some data that you can use in the comparison. It can be a pointer to data like that in the array, a pointer to another type of data, or NIL.

In THINK C, declare compare like this:

```
int compare (void *item1, void *item2);
```

In THINK Pascal, declare compare like this:

```
function compare (item1, item2: Ptr): integer;
```

Moving methods

MoveItemToIndex

```
procedure MoveItemToIndex (currentIndex, newIndex:
   longint);
void MoveItemToIndex (long currentIndex,
   long newIndex);
```

Move the element at currentIndex to newIndex. This method moves the elements between the old and new positions up or down slot. This method sends a BroadcastChange message with arrayMoveElement as the reason.

Swap

```
procedure Swap (index1, index2: longint);
void Swap (long index1, long index2);
```

Swap the two items: move the item at index1 to index2, and move the item at index2 to index1. For both items, this method sends a BroadcastChange message with arrayElementChanged as the reason.

Resizing methods

CArray uses these internal methods to resize its block of memory. You should need to use them only if you are creating a subclass of CArray. In THINK C, only subclasses of CArray can use them.

Resize

```
procedure Resize (numSlots: longint);
void Resize (long numSlots);
```

Resize the array so that it has `numSlots` slots. This method does not change the number of items stored in the array.

MoreSlots

```
procedure MoreSlots;
void MoreSlots (void);
```

Add `blockSize` slots to the array.

Temporary storage methods

CArray uses these methods to move elements into and out of the temporary storage slot. You should need to use them only if you are creating a subclass of CArray. In THINK C, only subclasses of CArray can use them

CopyToTemporary

```
procedure CopyToTemporary (index: longint);
void CopyToTemporary (long index);
```

Place the element in slot `index` into temporary storage. This method also sets `usingTemporary` to TRUE.

CopyFromTemporary

```
procedure CopyFromTemporary (index: longint);
void CopyFromTemporary (long index);
```

Place an item from temporary storage into slot `index`. This method also sets `usingTemporary` to FALSE.

Offset method

CArray uses this method to find the offset into the array of an element. You should need to use it only if you are creating a subclass of CArray. In THINK C, only subclasses of CArray can use it.

ItemOffset

```
function ItemOffset (itemIndex: longint): longint;
long ItemOffset (long itemIndex);
```

Return the offset into the array of the slot `itemIndex` in bytes.

CBartender 13

Introduction

The bartender is the object that manages the menu bar, menus, and menu items. The bartender is an object of class CBartender. There is only one object of this class stored in a global variable.

Note

Earlier versions of the THINK Class Library used CBarOwner, a subclass of CBartender, to prevent excessive menu bar flashing. This version of CBartender includes this improvement.

Heritage

Superclass	CObject
Subclasses	This class has no subclasses.

Using CBartender

In the THINK Class Library, almost every menu item has a command number associated with it. The bartender maintains a table that maps menu items with command numbers. You should never try to access the bartender's data structures directly. Instead, use the access methods.

You should not need to subclass the CBartender class. If, for some reason, you do need to create a subclass, you'll also need to override the `SetUpMenus` method in your application class to initialize your bartender.

To manipulate the menus or menu items, send messages to the global bartender object stored in the global variable `gBartender`.

Creating standard menus

To associate a command number with a menu item, append the command number to the end of the menu item in your `MENU` resource. The menu item

text and the command number are separated by the character #. Figure 13-1 shows what the **File** menu looks like in ResEdit:

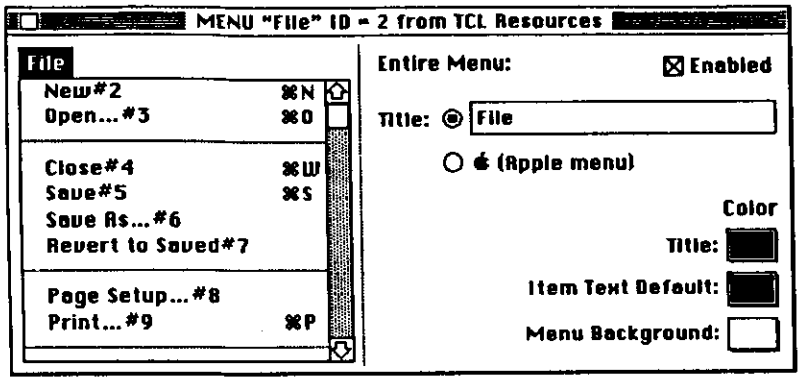

Figure 13-1 The File menu in ResEdit

If you don't append a command number to a menu item, the bartender automatically associates the command number `cmdNull` with it.

The bartender builds its tables from the information in the `MBAR` resource you pass to its initialization method. Your application's `MBAR` resource should contain the menu IDs of all the menus that will appear in the menu bar.

Note

By default, the application's `SetUpMenus` method uses the `MBAR` resource with an ID = 1.

The `MENU` resources for these menus are in the file `TCL Resources`.

You can use any menu ID for your application's menus. The THINK Class Library reserves the following menu IDs for certain menus:

Menu title	Menu ID	Mnemonic
Apple	1	`MENUapple`
File	2	`MENUfile`
Edit	3	`MENUedit`
Font	10	`MENUfont`
Size	11	`MENUsize`

These are the commands that the THINK Class Library defines. In THINK Pascal they're in the `TCL.p`, and in THINK C they're in the file `Commands.h`. Remember that the THINK Class Library reserves the commands in the range 1 to 1023.

These are standard **File** menu commands:

Command	ID
`cmdQuit`	1
`cmdNew`	2
`cmdOpen`	3
`cmdClose`	4
`cmdSave`	5
`cmdSaveAs`	6
`cmdRevert`	7
`cmdPageSetup`	8
`cmdPrint`	9

These are the standard **Edit** menu commands:

Command	ID
`cmdUndo`	16
`cmdCut`	18
`cmdCopy`	19
`cmdPaste`	20
`cmdClear`	21
`cmdToggleClip`	22
`cmdSelectAll`	23

These are the text style commands:

Command	ID
`cmdPlain`	30
`cmdBold`	31
`cmdItalic`	32
`cmdUnderline`	33
`cmdOutline`	34
`cmdShadow`	35
`cmdCondense`	36
`cmdExtend`	37

These are the text alignment commands:

Command	ID
cmdAlignRight	40
cmdAlignLeft	41
cmdAlignCenter	42
cmdJustify	43

These are the line spacing commands:

Command	ID
cmdSingleSpace	50
cmd1HalfSpace	51
cmdDoubleSpace	52

These are miscellaneous commands:

Command	ID	Description
cmdNull	0	Command which does nothing
cmdOK	100	OK button in dialog box
cmdCancel	101	Cancel button in dialog box
cmdAbout	256	About Application request

If you want the bartender to return the menu ID and item number of a particular menu item, use the special command -1 in your MENU resource. The bartender will return the negative of the menu ID in the high word and the menu item number in the low word. This is the same as menus that you build on the fly, described next.

Resource based menus

Your application may use menus that don't have menu commands associated with each item. The most common example is a **Font** menu. Use the reserved **Font** menu in your MBAR resource, and add the resources with the AddResMenu Toolbox routine. Here's how you might write your SetUpMenus method in THINK Pascal to include a font menu.

```
procedure CYourApp.SetUpMenus;
begin
   inherited SetUpMenus;
   AddResMenu(GetMHandle(MENUfont), 'FONT');
   SetDimOption(MENUfont, dimNONE);
   SetUnchecking(MENUfont, true);
end;
```


And here's how you might write your the method in THINK C:

```
void CYourApp::SetUpMenus()
{
   MenuHandle macMenu;

   inherited::SetUpMenus();
   AddResMenu(GetMHandle(MENUfont), 'FONT');
   SetDimOption(MENUfont, dimNONE);
   SetUnchecking(MENUfont, TRUE);
}
```

Note

To learn about SetDimOption and SetUnchecking, see section "Dimming and checking menu items" on page 197.

When you build a menu on the fly like this, the bartender does not associate a command number with the menu items. In this case, the bartender's FindCmdNumber returns the negative of the menu ID in the high word and the item ID in the low word.

The values that FindCmd *returns in these cases are the negative of what* MenuSelect *would have returned*

For example, if you choose the ninth font in the **Font** menu, FindCmdNumber returns -655369 (0xFFF5FFF7). Since the number is negative, you know that there is no command number associated with the item. To get the menu ID and the menu item, negate the value and split it into two words. In this example, the return value becomes 655369 (0x000A0009), which means that the menu ID is 10 and the menu item is 9.

Figure 13-2 How FindCmdNumber builds command numbers.

Note

There are functions to extract the high word and low word of a long integer. In THINK Pascal, the functions are HiWord and LoWord. In THINK C, the functions are HiShort and LoShort (defined in Global.h).

You can add menu items to existing menus if you like. You might want to add the **Font** menu to a general text-handling menu, or you might want to have a menu with the names of all the documents your application has

opened. The important thing to remember is to add all these menu items at the **end** of the existing menu. Otherwise, the bartender will get confused.

Creating hierarchical menus

For a detailed description of hierarchical menus, see Inside Macintosh V, *Chapter 13, "The Menu Manager."*

Most of the time you'll be able to set up your hierarchical menus in your resource file. The key thing to remember is that the item that the hierarchical menu is attached to uses the command key equivalent and the item mark differently. The command key value must be `0x1B` (Control-[), and the value of the item mark is the resource ID of the hierarchical menu.

Sometimes, you can't specify a hierarchical menu in a resource. In these cases, use the `InsertHierMenu` method. This method adds the hierarchical menu to an existing menu in the menu bar and to the bartender's table. Generally, you'll specify hierarchical menus in your resource file. Use this method to insert hierarchical menus from your program.

Note

For hierarchical menus to work correctly, you must set them up either in the resource file or with a call to `InsertHierMenu`. If you use only the Macintosh Toolbox routines, the bartender's item checking methods will not work.

This example shows how to write a method in THINK Pascal to insert a hierarchical **Font** menu as the third item in a **Text** menu from your program:

```
procedure CYourApp.SetUpMenus;
var
   macMenu: MenuHandle;
   MENUtext, itemNo: integer;
begin
   inherited SetUpMenus;
   gBartender.InsertHierMenu(MENUfont, cmdNull,
      MENUtext, 2);
   AddResMenu(GetMHandle(MENUfont), 'FONT');
   gBartender.SetDimOption(MENUfont, dimNONE);
   gBartender.SetUnchecking(MENUfont, true);
end;
```


And this example shows how to write a method in THINK C to do the same thing:

```
void CYourApp::SetUpMenus()
{
   MenuHandle  macMenu;
   short       MENUtext, itemNO;

   inherited::SetUpMenus();
   gBartender->InsertHierMenu(MENUfont, cmdNull,
      MENUtext, 2);
   AddResMenu(GetMHandle(MENUfont), 'FONT');
   SetDimOption(MENUfont, dimNONE);
   SetUnchecking(MENUfont, TRUE);
}
```

Dimming and checking menu items

The dimming, undimming, and unchecking take very little time. You won't notice a delay between the time you click on the menu bar and when the menu is displayed.

The bartender includes methods to let you enable and disable and check and uncheck menu items. When you click in the menu bar, the bartender sends an `UpdateMenus` message to all of the bureaucrats in the Chain of Command. In the general case, all the items in the menu start out dimmed and unchecked. Then each bureaucrat enables the menu items that pertain to it. Once the appropriate items have been enabled and checked, the Toolbox routine `MenuSelect` displays all the menus.

Suppose you click in the menu bar of a text processing application. When you click on the menu bar, but before the Toolbox displays the menu, the bartender disables all the menu items. Then, the application enables all the application related menu items: **New**, **Open...**, **Quit**, for example. The document enables all the document related items: **Save**, **Save As...**, **Revert** (if the document's been changed), and so on. A pane might check the current font and size in the **Font** menu. Finally the menu appears on the screen with the correct items checked and enabled.

The `UpdateMenus` method of your application, document, and pane need to enable each item. To make sure that item enabling happens from the general (application) to the specific (pane), be sure to call `inherited UpdateMenus` first in your own `UpdateMenus` method.

You can use the bureaucrat methods `SetDimOption` and `SetUnchecking` in your application `SetUpMenus` method to modify this behavior. `SetDimOption` lets you specify whether the bartender should dim all, some, or none of the items when you click on the menu bar. For **Font** menus, for instance, it doesn't make sense to dim all the font names only to re-enable them again.

Variables

The global variable `gBartender` points to the single instance of the bartender. Whenever you want to manipulate menus, you'll send messages to this object. The bartender uses its instance variables to maintain the mapping between command numbers and menu items. You should not need to access or alter the instance variables.

Global variable

Variable	Type	Description
`gBartender`	`CBartender`	The global bartender.

Instance variables

Variable	Type	Description
`numMenus`	`integer`	The number of menus available.
`theMenus`	`MenuEntryH`	A table with an entry for each available menu.
`choreAssigned`	`Boolean`	TRUE if there's a pending chore to redraw the menu bar
`forceMBarUpdate`	`Boolean`	TRUE if `UpdateMenuBar` always draws the menu bar.

Methods

To use these methods to manipulate menus and menu items, send messages to the bartender object stored in the global `gBartender`.

For example, to disable the **Open...** command you would write, in THINK Pascal:

```
gBartender.DisableCmd(cmdOpen);
```

And in THINK C you would write:

```
gBartender->DisableCmd(cmdOpen);
```

To change the name of the **Copy** command to **Copy Picture**, you might write, in THINK Pascal:

```
gBartender.SetCmdText(cmdCopy, 'Copy Picture');
```


And in THINK C you might write:

```
gBartender->SetCmdText(cmdCopy,
   "\pCopy Picture");
```

In the rare case where you need the actual menu handle or other information about the menu, use the `FindMacMenu` or `FindMenuItem` methods.

Construction methods

IBartender

```
procedure IBartender (MBARid: integer);
void IBartender (short MBARid);
```

Initialize a Bartender object. The application method `SetUpMenus` sends this message to initialize the bartender and stores the bartender in the global variable `gBartender`. `MBARid` is the ID of the `MBAR` resource that the bartender uses to build its menu tables. The default `SetUpMenus` method uses 1 as its `MBARid`.

Insertion and deletion methods

AddMenu

```
procedure AddMenu (MENUid: integer;
   install: Boolean; beforeID: integer);
void AddMenu (short MENUid, Boolean install,
   short beforeID);
```

This method adds a menu to the bartender's list. `MENUid` is the resource ID of the menu. If `install` is `TRUE`, install it into the menu bar as well as into the bartender's list. `BeforeID` specifies before which menu this menu is installed. If `beforeID` is 0, the menu is added to the end of the menu bar. A `beforeID` of -1 is a hierarchical or pop-up menu.

RemoveMenu

```
procedure RemoveMenu (MENUid: integer);
void RemoveMenu (short MENUid);
```

Remove the specified menu from the menu bar and from the bartender's list. `MENUid` is the resource ID of the menu.

InsertInBar

```
procedure InsertInBar (MENUid, beforeID: integer);
void InsertInBar (short MENUid, short beforeID);
```

Insert a menu in the menu bar and in the bartender's list. This message is the same as `AddMenu(MENUid, TRUE, beforeID)`.

DeleteFromBar

```
procedure DeleteFromBar (MENUid: integer);
void DeleteFromBar (short MENUid);
```

Remove the specified menu from the menu bar, but leave it in the list. After deleting the menu, this method creates a chore of class `CMBarChore` and

sends it to gApplication as an urgent chore. The chore will redraw the menu bar the next time through the event loop.

InsertHierMenu

```
procedure InsertHierMenu (hMENUid: integer;
   cmdNo: longint; inMENUid, afterItem: integer);
void InsertHierMenu (short hMENUid, long cmdNo,
   short inMENUid, short afterItem);
```

Insert a hierarchical menu in another menu. `HMENUid` is the menu ID of the hierarchical menu. If you want to be able to dim a hierarchical menu, provide a command number in `cmdNo`; if you're not going to dim the hierarchical menu, pass in `cmdNull` for `cmdNo`. `InMENUid` is the menu ID of the menu that will contain the hierarchical menu. `AfterItem` is the item number after which the hierarchical menu should be inserted. The title of the menu is used as the item name for the hierarchical menu.

Item manipulation methods

EnableCmd

```
procedure EnableCmd (cmdNo: longint);
void EnableCmd (long cmdNo);
```

Enable the menu item associated with `cmdNo`.

DisableCmd

```
procedure DisableCmd (cmdNo: longint);
void DisableCmd (long cmdNo);
```

Disable the menu item associated with `cmdNo`.

EnableMenu

```
procedure EnableMenu (MENUid: integer);
void EnableMenu (short MENUid);
```

Enable the menu with resource ID `MENUid`.

DisableMenu

```
procedure DisableMenu (MENUid: integer);
void DisableMenu (short MENUid);
```

Disable the menu with resource ID `MENUid`.

EnableMenuBar

```
procedure EnableMenuBar;
void EnableMenuBar (void);
```

Enable all the menus currently in the menu bar.

DisableMenuBar

```
procedure DisableMenuBar;
void DisableMenuBar (void);
```

Disable all the menus currently in the menu bar.

SetCmdText

```
procedure SetCmdText (cmdNo: longint;
   theText: Str255);
void SetCmdText (long cmdNo, Str255 theText);
```

Change the text of the menu item associated with `cmdNo` to `theText`.

GetCmdText

```
procedure GetCmdText (cmdNo: longint;
   var theText: Str255);
void GetCmdText (long cmdNo, Str255 theText);
```

Get the text of the menu item associated with `cmdNo`. If `cmdNo` is not associated with a menu item, the contents of `theText` are unchanged.

CheckMarkCmd

```
procedure CheckMarkCmd (cmdNo: longint;
   checked: Boolean);
void CheckMarkCmd (long cmdNo, Boolean checked);
```

Place (or remove) a check mark next to the menu item associated with `cmdNo`.

Item insertion and deletion

InsertMenuCmd

```
procedure InsertMenuCmd (cmdNo: longint;
   theText: Str255; MENUid, afterItem: integer);
void InsertMenuCmd (long cmdNo, Str255 theText,
   short MENUid, short afterItem);
```

Insert a new item in a menu. `CmdNo` is the command number of the new item. `TheText` is the text of the new item. `MENUid` is the menu ID you're inserting the new item into. `AfterItem` specifies after which item to insert the new item.

RemoveMenuCmd

```
procedure RemoveMenuCmd (cmdNo: longint);
void RemoveMenuCmd (long cmdNo);
```

Remove the menu item associated with the command `cmdNo` from its menu.

Look-up methods

These methods let you know which command number is associated with which menu item and which menu contains the menu item associated with a particular command number.

FindCmdNumber

```
function FindCmdNumber (MENUid, itemNo: integer):
   longint;
long FindCmdNumber (short MENUid, short itemNo);
```

Return the command number associated with the menu `MENUid` and `itemNo`. If the item has no command associated with it, this method returns `-MENUid` in the high word and `itemNo` in the low word.

This is the message that the desktop and switchboard send to the bartender to convert a menu selection or a menu key into a command number. Some menu items, like font menus, don't have commands associated with them. For these menus, `FindCmdNumber` returns the menu id and item number.

FindMenuItem

```
procedure FindMenuItem (cmdNo: longint;
   var MENUid: integer; var macMenu: MenuHandle;
   var itemNo: integer);
void FindMenuItem (long cmdNo, short *MENUid,
   MenuHandle *macMenu, short *itemNo);
```

Given a command number, return the menu ID, the handle to the menu, and the item number. If the command number isn't associated with a menu item, all three items are set to zero.

FindItemText

```
function FindItemText; (MENUid: integer;
   itemStr: Str255): integer;
short FindItemText (short MENUid, Str255 itemStr);
```

Return the item number in `MENUid` with the specified text. Return 0 if the menu doesn't have an item called `itemStr`. This method is useful for keeping track of items that don't have command numbers associated with them.

FindMacMenu

```
function FindMacMenu (MENUid: integer): MenuHandle;
MenuHandle FindMacMenu (short MENUid);
```

Return a handle to the Macintosh menu with ID `MENUid`. If the menu is not in the bartender's table, return `NIL`.

FindMenuIndex

```
function FindMenuIndex (MENUid: integer): integer;
short FindMenuIndex (short MENUid);
```

Return the index of the menu `MENUid` in the bartender's table. This is an internal method. You should not use this method.

Appearance methods

SetDimOption

```
procedure SetDimOption (MENUid: integer;
   aDimming: DimOption);
void SetDimOption (short MENUid, DimOption aDimming);
```

Set the dim option for the specified menu. By default, all the items in the menu will be dimmed when you click in the menu bar. The `UpdateMenus` method of each of your bureaucrats needs to enable the appropriate items.

Dim Option	Meaning
`dimNONE`	Never dim any of the menu items.
`dimSOME`	Dim only the menu items that have command numbers associated with them.
`dimALL`	Dim all of the menu items. Each bureaucrat's `UpdateMenus` method must enable the items for the commands it handles. This is the default.

SetUnchecking

```
procedure SetUnchecking (MENUid: integer,
   anUnchecking: Boolean);
void SetUnchecking (short MENUid,
   Boolean anUnchecking);
```

Set the checking option for the specified menu. By default, none of the items are unchecked when you click in the menu bar. If you set this option to `TRUE`, all the items will be unchecked, and your `UpdateMenus` methods must check the appropriate menu items.

If anUnchecking is...	Do this...
`TRUE`	Uncheck all the menu items at menu selection. Your `UpdateMenus` method should check the appropriate items. Set this option for menus like **Font** menus or **Style** menus.
`FALSE`	Don't uncheck any menu items at menu selection. This is the default since most menu items never need to be checked.

UpdateAllMenus

```
procedure UpdateAllMenus;
void UpdateAllMenus (void);
```

The bartender gets this message before menu selection. First it disables and unchecks all the items in the menus according to the dimming and unchecking options. Then it sends an `UpdateMenus` message to the gopher to enable the appropriate menu items. You should not use or override this method.

UpdateMenuBar

```
procedure UpdateMenuBar;
void UpdateMenuBar (void);
```

If the menu bar needs to be redrawn, call `DrawMenuBar`. The menu bar needs to be redrawn if the `forceMBarUpdate` flag is TRUE, or if the enabled state of any menu in the menu bar doesn't match the last recorded state.

Command Extraction methods

These three methods are internal methods that the bartender uses to find commands associated with menu items and to build its internal data structures. You should not use these methods. These are the THINK Pascal methods:

```
procedure ExtractCommands: (var theEntry: MenuEntry);
procedure ParseItemString (var itemStr: Str255;
   var cmdNo: longint);
procedure ExtractHierMenus (macMenu: MenuHandle;
   index: integer);
```

And these are the THINK C methods:

```
void ExtractCommands (MenuEntryP theEntry);
void ParseItemString (Str255 itemStr, long *cmdNo);
void ExtractHierMenus (MenuHandle macMenu,
   short index);
```


CBitMap 14

Introduction

CBitMap is a class for working with Macintosh bit maps. You should be familiar with QuickDraw bit maps to get the most out of this class.

Heritage

Superclass	CObject
Subclasses	None

Using CBitMap

CBitMap gives you methods to draw into a bit map, to stamp a bit map on the current drawing port, and to get a bit map from the current drawing port. You may notice that this class doesn't provide a drawing method. That's because this class is only for *manipulating* bit maps. The class CBitMapPane gives you a pane for drawing bit maps.

To learn more about transfer modes see Inside Macintosh I, *Chapter 6, "QuickDraw."*

Use the `CopyFrom` method to copy bits from your bit map object to the bit map of the current QuickDraw port. To copy bits from the current port to your bit map object, use the `CopyTo` method. The transfer mode determines how the bit mapped is copied. Use the `GetXferMode` method to examine the current transfer mode and the `SetXferMode` method to set the transfer mode. The default transfer mode is `srcCopy`

To draw into an off-screen bit map, bracket your drawing routines with calls to `BeginDrawing` and `EndDrawing`. When you create the bit map object, you have the choice of creating a drawing port for it or not. If you choose to create a port, your bit map will have a complete QuickDraw environment when you draw to it. If you don't, the drawing routines use the drawing environment of the current port.

In general, you should create a port for a bit map whenever you intend to draw to a bit map. If you use a bit map only to hold an image that you copied from or want to copy to the current port, you don't need to create a port.

For example, this Pascal procedure draws a diagonal line in an off-screen bit map.:

```
procedure DrawIntoOffScreenBitMap;
var
   myBitMap: CBitMap;
begin
   new(myBitMap);
   myBitMap.IBitMap(300, 300, TRUE);
   myBitMap.BeginDrawing;
   MoveTo(10, 10);
   LineTo(200, 200);
   myBitMap.EndDrawing;
end;
```

This is what the same routine looks like in C:

```
void DrawIntoOffScreenBitMap(void)
{
   CBitMap *myBitMap;

   myBitMap = new(CBitMap);
   myBitMap->IBitMap(300, 300, TRUE);
   myBitMap->BeginDrawing;
   MoveTo(10, 10);
   LineTo(200, 200);
   myBitMap->EndDrawing;
};
```

Once you finish drawing into your bit map, you can use the CopyFrom method to stamp it into the curent port. To save the bit map as a paint (PNTG) file, you can use the CPNTGFile class.

Variables

Variable	Type	Desktop
macPort	GrafPtr	Grafport for the bit map if requested.
savePort	GrafPtr	Used internally to save the original drawing port if necessary.
macBitMap	BitMap	The bit map.
saveBitMap	BitMap	Used internally to save the original drawing port's bit map if necessary.
xferMode	Integer	Bitmap transfer mode. Initially srcCopy.

Methods

Construction and destruction methods

IBitMap

```
procedure IBitMap (width, height: Integer;
   makePort: Boolean);
void IBitMap (short width, short height,
   Boolean makePort);
```

Initialize the bit map. This method allocates memory for the bit map. The origin is set to (0, 0), and the transfer mode is set to srcCopy. If makePort is TRUE, IBitMap creates a QuickDraw grafport whose port bits point to the bit map. In general, you should pass TRUE for makePort if you intend to draw to the bit map and FALSE if you use the bit map only to hold an image.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the bit map. This method releases memory allocated to the bit map and closes the macPort if it was opened.

Accessing methods

GetBounds

```
procedure GetBounds (var theBounds: LongRect);
void GetBounds (theBounds *LongRect);
```

Get the bounding rectangle of the bit map. This rectangle describes the size and coordinate system of the bit map.

SetBoundsOrigin

```
procedure SetBoundsOrigin (hOrigin, vOrigin: integer);
void SetBoundsOrigin (short hOrigin, short vOrigin);
```

Set the coordinates of the top left corner of the bit map. This method changes the coordinate system of the bit map.

SetXferMode

```
procedure SetXferMode (aXferMode: Integer);
void SetXferMode (short aXferMode);
```

Set the transfer mode of the bitmap.

GetXferMode

```
function GetXferMode: Integer;
short GetXferMode (void);
```

Return the transfer mode of the bit map.

PixelIsBlack

```
function PixelIsBlack (pixelPos: LongPt): Boolean;
Boolean PixelIsBlack (LongPt pixelPos);
```

Return TRUE if the pixel specified by pixelPos is black. Return FALSE if it isn't or if the specified position is not in the bit map.

Image copying methods

CopyTo

```
procedure CopyTo (fromRect, toRect: LongRect;
   maskRgn: RgnHandle);
void CopyTo (LongRect *fromRect, LongRect *toRect,
   RgnHandle maskRgn);
```

Copy bits to this bit map from the current port (`thePort`). `FromRect` is a rectangle in `thePort`'s bit map, and `toRect` is a rectangle in this bit map. If `maskRgn` is `NIL`, this method does no clipping. Otherwise, the transfer is clipped to `maskRgn`, a region in this bit map's coordinates. The transfer mode is stored in the `xferMode` instance variable

CopyFrom

```
procedure CopyFrom (LongRect, LongRect: Rect;
   maskRgn: RgnHandle);
void CopyFrom (LongRect *fromRect, LongRect *toRect,
   RgnHandle maskRgn);
```

Copy bits from this bit map to the current port (`thePort`). `FromRect` is a rectangle in this bit map, and `toRect` is a rectangle in `thePort`'s bit map. If `maskRgn` is `NIL`, this method does no clipping. Otherwise, the transfer is clipped to `maskRgn`, a region in `thePort`'s coordinates. The transfer mode is stored in the `xferMode` instance variable

Drawing preparation methods

BeginDrawing

```
procedure BeginDrawing;
void BeginDrawing (void);
```

Prepare for drawing to this bit map. It the bit map doesn't have its own port, set the port bits of the current port to this bit map. If the bit map has its own port, this method saves the current port and sets the bit map's port as the current port.

EndDrawing

```
procedure EndDrawing;
void EndDrawing (void);
```

Reset the port to the state before drawing. If the bit map doesn't have its own port, restore `thePort`'s port bits from `saveBitMap`, otherwise reset the port to `savePort`.

CBitMapPane 15

Introduction

CBitMapPane is a subclass of CPanorama for drawing bit maps in a pane.

Heritage

Superclass	CPanorama
Subclasses	None

Using CBitMapPane

CBitMapPane is a pane with an associated bit map. The drawing method of a bit map pane copies from the bit map to the pane's drawing port. That means that your program should draw into the bit map pane's bit map before the drawing method gets called. The Art Class demonstration program, which uses a subclass of CBitMapPane, draws directly on the screen and on the associated bit map in a mouse task. When the Macintosh generates an update event, the pane's drawing method copies the associated bit map right on the screen, giving a very smooth update.

For static images, a picture pane is easier to use. See the CPicture class on page 209.

Though you can use CBitMapPane directly, you will probably want to use a subclass of it so you can add instance variables and methods tailored for your application. Your program can use whatever technique is appropriate to fill the bit map with an image. If the image doesn't change, your program can just read the bit map from a file and rely on the update event to stamp the image on the screen. If it's an interactive image, you might use a mouse task like Art Class does. If it's a complex image and you want to give the illusion of a quick update, you do the drawing to the bit map in your subclass's `Draw` method with CBitMap's `BeginDrawing` and `EndDrawing` methods, then call the inherited `Draw` method do stamp the image on the screen.

Variables

Variable	Type	Desktop
itsBitMap	CBitMap	The bitmap to stamp on the pane.

Methods

Construction and destruction methods

IBitMapPane

```
procedure IBitMapPane (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption;
   aBounds: LongRect;
   aBitMap: CBitMap; makePort: Boolean);
```

```
void IBitMapPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   LongRect *aBounds,
   CBitMap *aBitMap, Boolean makePort);
```

Initialize a panorama. The first eight arguments to this routine are identica to the ones for IPane.

The aBounds rectangle is a long rectangle that defines the bounds of the panorama. The bounds define the height and width of aBitMap as well a its coordinate system.

If aBitMap is NIL, IBitMapPane creates a new bit map object for the pane, otherwise it uses aBitMap as the pane's bit map. When you use an existing bit map, be sure to get pass the correct bounds to IBitMapPane Send the bit map a GetBounds message to get its bounds, and pass the r sults to IBitMapPane.

IBitMapPane passes the makePort parameter to CBitMap's IBitMap method. If aBitMap is not NIL, this parameter is ignored. The makePor parameter specifies whether the bit map should have a drawing environment. In general, you should always pass TRUE for this parameter.

Note

The descriptions of the other arguments are in CPane on page 347.

Free/Dispose

```
procedure Free;
```

```
void Dispose (void);
```

Dispose of the bit map. This method releases memory allocated to the bit map and closes the macPort if it was opened.

Accessing methods

SetBitMap

```
procedure SetBitMap (aBitMap: CBitMap);
```

```
void SetBitMap (CBitMap *aBitMap);
```

Get the bounding rectangle of the bit map. This rectangle describes the size and coordinate system of the bit map.

GetBitMap

```
function GetBitMap: CBitMap;
```

```
CBitMap *GetBitMap (void);
```

Get the bit map associated with this pane.

Drawing method

Draw

```
procedure Draw (var area: Rect);
```

```
void Draw (Rect *area);
```

Draw the bit map. Copy the bits in the bit map to the bit map of the current port.

CBureaucrat 16

Introduction

CBureaucrat is an abstract class that implements a link in the chain of command. Any object in the THINK Class Library that responds to a menu command or a mouse click is a bureaucrat.

A bureaucrat can either respond to a command, or it can pass the command to its supervisor. All of the default methods for bureaucrats pass the message on to the supervisor, so if a particular subclass cannot handle the message, calling the inherited method will cause the message to be passed on up the chain of command.

Heritage

Superclass	CCollaborator
Subclasses	CDirectorOwner
	CView

Using CBureaucrat

You generally won't need to create a subclass of CBureaucrat because most of the objects you'll use are already descendants of this class. However, you may find that you need to create a class that's a link in the chain of command that's not one of the common objects.

The global variable `gGopher` points to the current bureaucrat which is the first object in the chain of command. Whenever the switchboard responds to an event, it sends an appropriate message to the gopher. For instance, in response to a key-down event, the switchboard sends a `DoKeyDown` message to the gopher.

If the object `gGopher` refers to cannot handle the command, it passes the message up the chain of command until the message reaches an object that handles the command or until it reaches the end of the chain of command—the application.

When there are no documents open, gGopher points to the application. When you open a document, gGopher points to the document. Your document subclass should use the BecomeGopher method to change gGopher to point to the main pane.

Be sure to use the BecomeGopher method to change the gopher instead of setting the gGopher variable yourself. BecomeGopher uses CCollaborator's provider/dependent mechanism to let dependent objects know that the gopher has changed.

Variables

Global variable

Variable	Type	Description
gGopher	CBureaucrat	The current bureaucrat.

Instance variable

A bureaucrat has only one instance variable.

Variable	Type	Description
itsSupervisor	CBureaucrat	The supervisor.

Methods

Construction and destruction methods

IBureaucrat

```
procedure IBureaucrat (aSupervisor: CBureaucrat);
void IBureaucrat (CBureaucrat *aSupervisor);
```

Initialize the bureaucrat. This method sets aSupervisor to be itsSupervisor

Free/Dispose

```
procedure Free;
void Dispose (void);
```

If the bureaucrat is the gopher, send a BecomeGopher(TRUE) message to the supervisor before disposing of the object.

Accessing method

GetSupervisor

```
function GetSupervisor: CBureaucrat;
CBureaucrat *GetSupervisor (void);
```

Return the bureaucrat's supervisor

Command methods

These are the messages that every bureaucrat accepts. Unless a bureaucrat (or one of its ancestors) overrides one of these methods, it just passes the message to its supervisor. If you follow the chain of supervisors—the chain of command—all the way to the end, you'll end up at your application class.

Notify

```
procedure Notify (theTask: CTask);
void Notify (CTask *theTask);
```

A task has been completed. This method passes the Notify message to the bureaucrat's supervisor.

DoKeyDown

```
procedure DoKeyDown (theChar: char; keyCode: Byte;
   macEvent: EventRecord);
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handle a key-down event. This method passes the DoKeyDown message to the bureaucrat's supervisor.

DoAutoKey

```
procedure DoAutoKey (theChar: char; keyCode: Byte;
   macEvent: EventRecord);
void DoAutoKey (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handle an auto-key event. This method passes the DoAutoKey message to the bureaucrat's supervisor.

DoKeyUp

```
procedure DoKeyUp (theChar: char; keyCode: Byte;
   macEvent: EventRecordPtr);
void DoKeyUp (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

Handle a key-up event. This method passes the DoKeyUp message to the bureaucrat's supervisor.

Note

By default, the Toolbox Event Manager masks out key-up events.

DoCommand

```
procedure DoCommand (theCommand: longint);
void DoCommand (long theCommand);
```

Handle a command. This method passes the DoCommand message to the bureaucrat's supervisor.

DoAppleEvent

```
procedure DoAppleEvent (anAppleEvent: CAppleEvent);
void DoAppleEvent (CAppleEvent *anAppleEvent);
```

Handle an AppleEvent. This method passes the AppleEvent to the bureaucrat's supervisor.

Dawdle

```
procedure Dawdle (var maxSleep: longint);
void Dawdle (long *maxSleep);
```

Perform periodic actions at idle time. The default method does nothing. CApplication's `Idle` method sends a `Dawdle` message to all of the bureaucrats in the chain of command.

If your bureaucrat requires periodic actions, perform them in the `Dawdle` method. Set the value of `maxSleep` to the largest number of ticks that your application can tolerate between `Dawdle` messages. If your application's `Dawdle` message doesn't have any time constraints, you can ignore `maxSleep`.

The application class uses the value of `maxSleep` to let `WaitNextEvent` know that it should yield an event after at most `maxSleep` ticks. For instance, the `Dawdle` method for CEditText calls `TEIdle` to blink the insertion point and sets `maxSleep` to `GetCaretTime` which is the rate at which the insertion point blinks.

A blinking insertion point is a good example. A clock display is another good example. You would update the display from your document's `Dawdle` method. If your clock displays seconds, set `maxSleep` to 60 since you need to update the display at least every 60 ticks.

On a null event, CApplication's `Idle` method sends a `Dawdle` message to every bureaucrat in the chain of command. `Idle` sets the global variable `gSleepTime` to the smallest sleep time that all bureaucrats requested. The switchboard uses this global variable in its call to `WaitNextEvent`.

UpdateMenus

```
procedure UpdateMenus;
void UpdateMenus (void);
```

Update the menus that have to do with this bureaucrat. Your bureaucrat (usually a document or a pane) should override this method to enable the menu items that pertain to it. The important thing to remember when you write an `UpdateMenus` method is to call `inherited UpdateMenus` before you enable your own menus. In this way, your supervisor's menus will be updated first. See CBartender, especially section "Dimming and checking menu items" on page 197, for a discussion of menu updating.

BecomeGopher

```
function BecomeGopher (fBecoming: Boolean): Boolean;
Boolean BecomeGopher (Boolean fBecoming);
```

If `fBecoming` is TRUE, make this bureaucrat the gopher and call `BroadcastChange` with `bureaucratIsGopher` as the reason. If `fBecoming` is FALSE, call `BroadcastChange` with `bureaucratIsNotGopher` as the reason.

If the current gopher refuses to relinquish control, this method returns FALSE and does not call `BroadcastChange` at all.

Change notification methods

These methods override methods in CCollaborator. To learn how to use them, see CCollaborator on page 249.

BroadcastChange

```
procedure BroadcastChange (reason: longint;
   info: Ptr);
void BroadcastChange (long reason, void* info);
```

In addition to notifying this bureaucrat's dependents of a change, notify its supervisor as well. `Reason` is an integer that describes the type of change. You must define reasons for your subclass. `Info` is a pointer to any additional information needed to respond to the change. If you want objects that aren't in a collaborator's list of dependents to know about a change, override this method

ProviderChanged

```
procedure ProviderChanged (aProvider: CCollaborator;
   reason: longint; info: Ptr);
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

One of this bureaucrat's providers has just changed. This method passes the change notification to the bureaucrat's supervisor. If you want your subclass to respond to change notifications, you should override this method. `aProvider` is the provider that changed. `Reason` is an integer that describes the type of change. You must define reasons for your subclass. `Info` is a pointer to any additional information that your subclass might need.

CButton 17

Introduction

CButton implements a standard Macintosh push button. You can use a button in any pane or window.

Heritage

Superclass	CControl
Subclasses	CCheckBox
	CRadioControl

Using CButton

CButton implements a standard Macintosh push button. You should not override this class unless you want to implement a push button that works differently from the default button. (The other kinds of Macintosh buttons, check boxes and radio buttons, are subclasses of this class.)

The CButton class handles all the mouse tracking for you. When you click and release the mouse button within a button, the button sends a `DoCommand` message to its supervisor. To specify which command number the button will send in the `DoCommand` message, send it a `SetClickCmd` message.

This is what happens when you click in a button. The `DoClick` method that CButton inherits from CControl calls the Macintosh Toolbox routine `TrackControl` to highlight the button and track mouse movement. When you release the mouse within a button, the `DoClick` method sends the control a `DoGoodClick` message. The `DoGoodClick` method sends the `DoCommand` message to the button's supervisor.

Variables

Variable	Type	Description
clickCmd	longint	Command number to send after a click. The default is cmdNull.

Methods

Although CButton implements only these methods, you can use the methods it inherits from CControl and CView to manipulate a button's title and location.

IButton

```
procedure IButton (CNTLid: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void IButton (short CNTLid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialize a button from a CNTL resource (a control template). CNTLid is the resource ID of the CNTL resource. AnEnclosure is the pane or window that the control belongs to. ASupervisor is the control's supervisor, typically a document.

IButton places the control at the location specified in the CNTL resource. If you want to position the button from your program, send it a Place message. (Controls inherit the Place method from CPane.)

INewButton

```
procedure INewButton (aWidth, aHeight: integer;
   aHEncl, aVEncl: integer; title: StringPtr;
   fVisible: Boolean; procID: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void INewButton (short aWidth, short aHeight,
   short aHEncl, short aVEncl, StringPtr title,
   Boolean fVisible, short procID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a button from the parameters in the argument list. Title is the name of the button. FVisible specifies whether the button is drawn initially. ProcID specifies what type of button to create. Its possible values are defined in Controls.h and include pushButProc, checkBoxProc, and radioButProc.

Note

The descriptions of the other arguments are in CPane on page 347.

GetClickCmd

```
function GetClickCmd: longint;
long GetClickCmd (void);
```

Return the command which is sent to a button's supervisor when the user clicks in a button.

SetClickCmd

```
procedure SetClickCmd (aClickCmd: longint);
void SetClickCmd (long aClickCmd);
```

Set the command number to be sent when the user clicks in a button.

SetDefault

```
procedure SetDefault (fDefault: Boolean);
void SetDefault (Boolean fDefault);
```

Specify whether the button is the default button. The default button has a three-pixel thick rounded rectangle as its border.

SimulateClick

```
procedure SimulateClick;
void SimulateClick (void);
```

Simulate a click in this button. Use this method when you want to provide visual feedback for a Command key shortcut. This method highlights the button momentarily, then sends a DoGoodClick message to the button.

DoGoodClick

```
procedure DoGoodClick (whichPart: integer);
void DoGoodClick (short whichPart);
```

When you press and release the mouse within the button, this method sends a DoCommand message to its supervisor. This is an internal method. You should not use or override this method.

CCharGrid 18

Introduction

CCharGrid is a subclass of CGridSelector that displays characters in a table and lets you choose one. CCharGrid is useful for implementing tool palettes like MacPaint and HyperCard.

Heritage

Superclass	CGridSelector
Subclasses	None

Using CCharGrid

The CCharGrid class lets you create panes that display characters in a table. The most common use for this kind of table is a tool palette. You can use a CCharGrid as a tool palette that's part of a window or as a custom tear-off menu. The Art Class demonstation program uses CCharGrid to display its Tool tear-off menu.

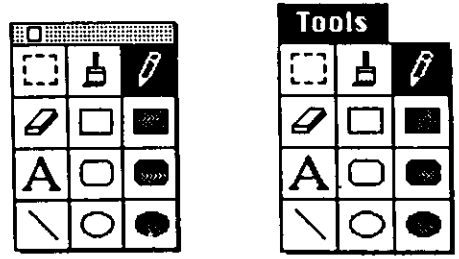

Figure 18-1 Art Class uses CCharGrid as a palette and as a menu

To use a CCharGrid, you need to create a special font that contains your tools. You can use ResEdit to create the font, or you can use a commercial a bit-map font editor. Apple recommends that the family ID for these kinds of fonts be in the range 32256 to 32767. Check with the Font Manager chapters

in *Inside Macintosh I* and *Inside Macintosh IV* for detailed information about working with fonts..

Unlike other subclasses of CPane that optionally let you initialize an object from a resource, you *must* use a resource to initialize a CCharGrid. The first argument to the initialization method, `ICharGrid`, is the resource ID of a ChGd resource which contains the values that `ICharGrid` passes up to CGridSelector's initialization method. The ChGd resource looks like this:

Field	Size	Description
Rows	integer	The number of rows in the grid.
Columns	integer	The number of columns in the gird.
Box Width	integer	The width in pixels of each box.
Box Height	integer	The height in pixels of each box.
Horiz. Sizing	integer	Horizontal sizing option. Usually `sizFIXEDLEFT` (0)
Vert. Sizing	integer	Vertical sizing option. Usually `sizFIXEDTOP` (2)
Horiz. Location	integer	Horizontal location of grid in its enclosure.
Vert. Location	integer	Vertical location of grid in its enclosure.
Command base	integer	The command base for turning selections into command numbers.
Font Size	integer	The size in points of the font.
Font Name	Pascal stirng	The name of the font to display.
theCharacters	Pascal string	Handle to a Pascal string of the characters to display in the grid.

If the first character of your font name is "%" or ".", it won't appear in your Font menu.

The first nine fields are the same as the arguments that you pass to `IGridSelector`. The last three are specific to character grids. The Font Size and Font Name fields specify the size and the name of the font to use. The font name is a Pascal string. The last field contains the characters to display. It is also a Pascal string.

The file `TCL TMPLs` contains ResEdit templates (`'TMPL'` resources) that help you create and edit `'ChGd'` resources. This is what the `'ChGd'` resource in Art Class looks like when you edit it with ResEdit.

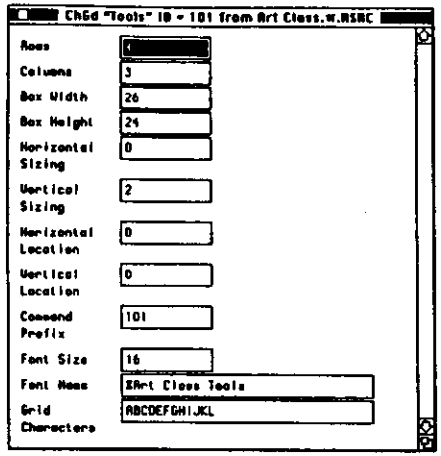

Figure 18-2 The ChGd resource in Art Class

Variables

Variable	Type	Description
`theCharacters`	`Handle`	Handle to a Pascal string of the characters to display in the grid.

Methods

Construction methods

ICharGrid

```
procedure ICharGrid. (ChGdid: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void ICharGrid (short ChGdid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialize the character grid. `ChGdid` is the resource ID of the `'ChGd'` resource that describes the location of the pane and the size and name of the font to use for the grid. See "Using CCharGrid" above for the details on `'ChGd'` resources. `ICharGrid` creates a text environment object of class CTextEnvirons to maintain the text characteristics whenever the characters are drawn. The text transfer mode is initially `srcOr`.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the character list and the object.

Drawing methods

DrawItem

```
procedure DrawItem (theItem: integer;
   theBox: Rect);
void DrawItem (short theItem, Rect *theBox);
```

Draw character numbered `theItem` so it is centered within the box. Characters are numbered from 1.

CCheckBox 19

Introduction

CCheckBox implements a standard Macintosh check box. You can use a check box in any pane or window.

Heritage

Superclass	CButton
Subclasses	None

Using CCheckBox

CCheckBox implements a standard Macintosh check box. You should not override this class unless you want to implement a check box that works differently from the default check box.

The CCheckBox class handles all the mouse tracking for you. When you click and release the mouse button within a check box, it toggles the value of the check box and sends a `DoCommand` message to its supervisor. To specify which command number the check box will send in the `DoCommand` message, send it a `SetClickCmd` message.

Note

CCheckBox inherits the `SetClickCmd` method from CButton.

This is what happens when you click in a check box. The `DoClick` method that CCheckBox inherits from CControl calls the Macintosh Toolbox routine `TrackControl` to highlight the check box and track mouse movement. When you release the mouse within a check box, the `DoClick` method sends the control a `DoGoodClick` message. The `DoGoodClick` method toggles the value of the control and sends the `DoCommand` message to the check box's supervisor.

To find out whether the check box is checked, send it an `IsChecked` message. You can set the value of the check box from your program by sending it a `SetValue` message. Use the values `BUTTON_ON` and `BUTTON_OFF` to specify whether to check or uncheck the check box.

Note

Setting the value with `SetValue` will not send a `DoCommand` message to the check box's supervisor.

Variables

This class has no instance variables.

Methods

Although CCheckBox implements only these methods you can use the methods it inherits from CControl and CView to manipulate a check box's title and location.

You should use the `SetClickCmd` method that CCheckBox inherits from CButton to specify the command the check box sends to its supervisor when you click in it.

ICheckBox

```
procedure ICheckBox (CNTLid: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void ICheckBox (short CNTLid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialize a check box from a `CNTL` resource (a control template). `CNTLid` is the resource ID of the `CNTL` resource `AnEnclosure` is the pane or window that the control belongs to. `ASupervisor` is the control's supervisor, typically a document.

`ICheckBox` places the control at the location specified in the `CNTL` resource. If you want to position the check box from your program, send it a `Place` message. (Controls inherit the `Place` method from CPane.)

INewCheckBox

```
procedure INewCheckBox (aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   title: StringPtr; fVisible: Boolean;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void INewCheckBox (short aWidth, short aHeight,
   short aHEncl, short aVEncl,
```



```
   StringPtr title, Boolean fVisible,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a check box from the parameters in the argument list. `Title` is the name of the check box. `FVisible` specifies whether the check box is drawn initially.

Note

The descriptions of the other arguments are in CPane on page 347.

DoGoodClick

```
procedure DoGoodClick (whichPart: integer);
void DoGoodClick (short whichPart);
```

When you press and release the mouse within the check box, this method toggles the state of the check box and sends a `DoCommand` message to its supervisor.

This is an internal method. You should not use or override this method.

IsChecked

```
function IsChecked: Boolean;
Boolean IsChecked (void);
```

Returns TRUE if the check box is checked.

CChore 20

Introduction

CChore is an abstract class for implementing periodic or urgent actions. The THINK Class Library executes periodic chores at idle time and urgent chores after processing the current event.

Heritage

Superclass	CObject
Subclasses	CMBarChore
	CTearChore

Using CChore

You can use this class to create chores that run at idle time or that run at the next possible opportunity. Your application maintains a list of chores and implements the methods to install and remove them.

To implement a chore, you need to define a subclass of this class. In your subclass, override the `Perform` method. This method implements the action you want your chore to take.

Idle chores

To install an idle chore, send it to the application in an `AssignIdleChore` message. Your application sends a `Perform` message to all the idle chores at idle time. To remove an idle chore, send the chore you want to remove to the application in a `CancelIdleChore` message. This example shows how to install an idle chore in THINK Pascal:

```
gApplication.AssignIdleChore(theChore);
```

And this example shows how to install an idle chore in THINK C:

```
gApplication->AssignIdleChore(theChore);
```

Note

To remove an idle chore, keep it in a variable since `CancelIdleChore` requires a reference to a CChore object.

Urgent chores

An urgent chore is one that gets executed immediately after the application processes the current event. Typically, it's the same event that caused the urgent chore to be installed. Urgent chores are executed only once. They're removed from the urgent chore list immediately after they're executed. To install an urgent chore, send it to the application in an `AssignUrgentChore` message.

Using chores

Chores are attached to the application and not to any particular document. The things that a chore does, then, should pertain to the entire application.

Note

If you need your chore to run at a specific interval, you can use the Macintosh Toolbox routine `Ticks` to store the last time the chore was performed.

For periodic tasks that apply to a particular document or pane, use the `Dawdle` method. All bureaucrats, including documents and panes, inherit a `Dawdle` method. The application sends a `Dawdle` message to each bureaucrat in the chain of command. For instance, you would use a `Dawdle` method to blink an insertion point in a pane.

Variables

This class has no instance variables. Your CChore subclass may define its own instance variables. For example, if you want your idle chore to run at specified intervals, you could use an instance variable to store the period.

Methods

CChore implements only one method, which you must override. Your CChore subclass may implement additional methods.

Perform

```
procedure Perform (var maxSleep: longint);
void Perform (long *maxSleep);
```

This is the method that executes your chore. You must override this method in your CChore subclass. The default method does nothing.

The `maxSleep` parameter specifies how many ticks your chore can tolerate passing before it gets a `Perform` message. For example, if you need to do a chore at least once every tenth of a second, you would set `maxSleep` to 6. (Each tick is one-sixtieth of a second.)

For a more detailed discussion of `maxSleep`, see the description of the `Dawdle` method in CBureaucrat on page 216.

CClipboard 21

Introduction

CClipboard implements a standard Macintosh clipboard, or scrap. The default clipboard handles `TEXT` and `PICT` scraps. To deal with other kinds of scraps or to implement a private scrap, create a subclass of this class.

Heritage

Superclass	CDirector
Subclasses	To implement a private scrap, you should define a subclass.

Using CClipboard

This class implements a standard Macintosh clipboard. CClipboard uses the desk scrap to store its data and supports a window to display the contents of the scrap. This implementation supports only `TEXT` and `PICT` data.

When you initialize your application, the default initialization method, `IApplication`, sends a `MakeClipboard` message. This method creates an instance of CClipboard and stores it in the global variable `gClipboard`.

The default application `DoCommand` method handles the `cmdToggleClip` command which shows and hides the Clipboard window. When you're running under MultiFinder, the Clipboard window is hidden when your application is suspended. When the application resumes, the Clipboard window becomes visible again.

Note

In the THINK Class Library, desk accessories behave as if they were in their own layer even if your application isn't running under MultiFinder. When a desk accessory window is frontmost, the Clipboard window will be hidden. When one of your application windows is frontmost, the Clipboard window will be visible again.

Implementing a CClipboard subclass

If you want your application to support a private scrap, or if you want to display other types of data, you'll need to implement a subclass of CClipboard. You'll also need to override the `MakeClipboard` method in your application class.

Note

If you create a subclass of this class, be sure you understand how the Macintosh scrap mechanism works. See *Inside Macintosh I*, Chapter 15, "The Scrap Manager."

In your CClipboard subclass you'll need to override these methods:

Method	Action
`PutData`	Place data in the private scrap.
`GetData`	Retrieve data from the private scrap.
`ConvertGlobal`	Place the contents of the desk scrap into the private scrap.
`ConvertPrivate`	Place the contents of the private scrap into the desk scrap.
`MakeClipView`	Create a view to display data other than plain text of pictures.

Variables

The global variable `gClipboard` contains a pointer to the single instance of the clipboard object. The value of this variable is set in the application method `MakeClipboard`.

Global variable

Variable	Type	Description
`gClipboard`	`CClipboard`	The global clipboard.

Instance variables

Variable	Type	Description
`itsContents`	`CPanorama`	Pane for displaying contents
`itsScrollPane`	`CScrollPane`	Contents can be scrolled
`theLength`	`longint`	Length from last Get operation
`theOffset`	`longint`	Offset from last Get operation
`lastScrapCount`	`integer`	Count at the last conversion between global and private scraps.
`privateNewer`	`Boolean`	`TRUE` if the private scrap has changed since the last conversion to the desk scrap.
`windowVisible`	`Boolean`	`TRUE` if the Clipboard window is visible.

Methods

Construction and destruction methods

IClipboard

```
procedure IClipboard (aSupervisor: CApplication;
   hasWindow: Boolean);
void IClipboard (CApplication *aSupervisor,
   Boolean hasWindow);
```

Initialize the clipboard. The application's `MakeClipboard` method sends this message and stores the clipboard object in the global variable `gClipboard`.

Note

If you create a subclass of CClipboard, be sure to override the `MakeClipboard` method in your application subclass as well so it creates an object of the clipboard subclass. See the implementation of `MakeClipboard` in CApplication for an example.

Suspend and resume methods

These methods handle scrap conversion when your application moves from foreground to background under MultiFinder. In most cases, you won't need to use or override these methods.

Suspend

```
procedure Suspend;
void Suspend (void);
```

The application is about to be switched into the background under MultiFinder. If the application's private scrap is newer than the desk scrap, this method sends `ConvertPrivate` and `ScrapConverted` messages to the clipboard. This method then sends the clipboard window a `HideSuspend` message.

Resume

```
procedure Resume;
void Resume (void);
```

The application is about to be brought to the foreground under MultiFinder. If the desk scrap is newer than the private scrap, this method sends the clipboard `ConvertGlobal`, `ScrapConverted`, and `UpdateDisplay` messages to update the contents of the clipboard. This method then sends the window a `ShowResume` message to make it visible.

Appearance methods

These methods handle the appearance of the Clipboard window and how the contents of the clipboard appear in the window. If you implement your own clipboard class, you'll need to override the `UpdateDisplay` method.

Close

```
function Close (quitting: Boolean): Boolean;
Boolean Close (Boolean quitting);
```

The user chose **Close** from the **File** menu. `Close` sends a `CloseWind` message and returns `TRUE`.

CloseWind

```
procedure CloseWind (theWindow: CWindow);
void CloseWind (CWindow *theWindow);
```

The user clicked in the window's close box. This method hides the Clipboard window and changes the text in the menu item.

Toggle

```
procedure Toggle;
void Toggle (void);
```

This method opens the Clipboard window if it's closed or closes it if it's open. The application's default `DoCommand` method sends this message.

UpdateDisplay

```
procedure UpdateDisplay;
void UpdateDisplay (void);
```

Display the contents of the scrap in the Clipboard window. This method supports only `TEXT` and `PICT` types of data. This method will work correctly with a private scrap, but if you want to display other kinds of data, you'll need to override this method.

Accessing methods

Use these methods to place data in and retrieve data from the desk scrap. If your application implements a private scrap, use the `PutData` and `GetData` methods instead.

PutGlobalScrap

```
procedure PutGlobalScrap (theType: ResType;
   theData: Handle);
void PutGlobalScrap (ResType theType, Handle theData);
```

Put `theData` of type `theType` into the desk scrap. Be sure to send an `EmptyGlobalScrap` message before you call this method.

GetGlobalScrap

```
function GetGlobalScrap (theType: ResType;
   theData: Handle): Boolean;
Boolean GetGlobalScrap (ResType theType,
   Handle theData);
```

Get the data of type `theType` from the desk scrap and put it in the block that `theData` is a handle to. Return `TRUE` if the method was able to fulfill the request, `FALSE` otherwise.

`TheData` must be an allocated handle. The size of the allocated memory will grow as needed to fit the data. This is how you would get the `TEXT` data from the desk scrap in THINK Pascal:

```
var
   myData: Handle;
   …
begin
      { Create a zero-sized handle }
   myData := NewHandle(0);
   gClipboard.GetGlobalData ('TEXT', myData);
   …
end
```

And this is how you would get TEXT data from the desk scrap in THINK C:

```
Handle myData;
…
   /* Create a zero-sized handle */
myData = NewHandle(0);
gClipboard->GetGlobalData('TEXT', myData);
```

taSize

```
function DataSize (theType: ResType): longint;
long DataSize (ResType theType);
```

Return the number of bytes of data in the clipboard of type `theType`. If the clipboard doesn't contain any data of the specified type, this method returns zero.

atus

```
function Status: ScrapStatus;
ScrapStatus Status (void);
```

Return the status of the scrap. This method returns a value that describes the relationship between the private scrap and the desk scrap.

Value	Meaning
`PRIVATE_SCRAP_NEWER`	The information in the private scrap is newer than the information in the desk scrap.
`GLOBAL_SCRAP_NEWER`	The information in the desk scrap is newer than the information in the private scrap.
`SCRAPS_THE_SAME`	The information in both scraps is the same.

crapConverted

```
procedure ScrapConverted;
void ScrapConverted (void);
```

Set the internal flags after the scrap has been converted.

Scrap conversion methods

Use the `PutData` and `GetData` methods to implement **Cut**, **Copy**, and **Paste** commands for your application. If your application uses only the desk scrap, you can use the `PutGlobalScrap` and `GetGlobalScrap` methods.

PutData

```
procedure PutData (theType: ResType; theData: Handle);
void PutData (ResType theType, Handle theData);
```

Put `theData` of type `theType` into the scrap. The default method puts the data in the desk scrap. If your subclass supports a private scrap, you must override this method. After you store your data in the private scrap, be sure to send a `PrivateChanged` message. Before you call this method, be sure you send an `EmptyScrap` method to clear your scrap.

GetData

```
function GetData (theType: ResType,
   var theData: Handle): Boolean;
Boolean GetData (ResType theType, Handle *theData);
```

Get the data of type `theType` and put it into a newly created handle. This method returns `TRUE` if the request was successful, `FALSE` otherwise. If your application supports a private scrap, you must override this method.

Unlike `GetGlobalScrap` you should not allocate the handle to `theData`. This method will allocate the memory. This is how you'd get data of type `TEXT` in THINK Pascal:

```
var
   Handle   myData;
begin
   gClipboard.GetData('TEXT', myData);
   …
end;
```

And this is how you'd get data of type `TEXT` in THINK C:

```
Handle myData;
…
gClipboard->GetData('TEXT', &myData);
```

ConvertGlobal

```
procedure ConvertGlobal;
void ConvertGlobal (void);
```

Convert data in the desk scrap and put it in the private scrap. The default method does nothing. If your application supports a private scrap, you must override this method.

ConvertPrivate

```
procedure ConvertPrivate;
void ConvertPrivate (void);
```

Convert data in the private scrap and put it into the desk scrap. The default method does nothing. If your application supports a private scrap, you must override this method.

EmptyGlobalScrap

```
procedure EmptyGlobalScrap;
void EmptyGlobalScrap (void)
```

Clear the contents of the desk scrap.

And this is how you would get TEXT data from the desk scrap in THINK C:

```
Handle myData;
…
   /* Create a zero-sized handle */
myData = NewHandle(0);
gClipboard->GetGlobalData('TEXT', myData);
```

DataSize

```
function DataSize (theType: ResType): longint;
long DataSize (ResType theType);
```

Return the number of bytes of data in the clipboard of type `theType`. If the clipboard doesn't contain any data of the specified type, this method returns zero.

Status

```
function Status: ScrapStatus;
ScrapStatus Status (void);
```

Return the status of the scrap. This method returns a value that describes the relationship between the private scrap and the desk scrap.

Value	Meaning
`PRIVATE_SCRAP_NEWER`	The information in the private scrap is newer than the information in the desk scrap.
`GLOBAL_SCRAP_NEWER`	The information in the desk scrap is newer than the information in the private scrap.
`SCRAPS_THE_SAME`	The information in both scraps is the same.

ScrapConverted

```
procedure ScrapConverted;
void ScrapConverted (void);
```

Set the internal flags after the scrap has been converted.

Scrap conversion methods

Use the `PutData` and `GetData` methods to implement **Cut**, **Copy**, and **Paste** commands for your application. If your application uses only the desk scrap, you can use the `PutGlobalScrap` and `GetGlobalScrap` methods.

PutData

```
procedure PutData (theType: ResType; theData: Handle);
void PutData (ResType theType, Handle theData);
```

Put `theData` of type `theType` into the scrap. The default method puts the data in the desk scrap. If your subclass supports a private scrap, you must

override this method. After you store your data in the private scrap, be sure to send a `PrivateChanged` message. Before you call this method, be sure you send an `EmptyScrap` method to clear your scrap.

GetData

```
function GetData (theType: ResType,
   var theData: Handle): Boolean;
Boolean GetData (ResType theType, Handle *theData);
```

Get the data of type `theType` and put it into a newly created handle. This method returns `TRUE` if the request was successful, `FALSE` otherwise. If your application supports a private scrap, you must override this method.

Unlike `GetGlobalScrap` you should not allocate the handle to `theData`. This method will allocate the memory. This is how you'd get data of type `TEXT` in THINK Pascal:

```
var
   Handle   myData;
begin
   gClipboard.GetData('TEXT', myData);
   …
end;
```

And this is how you'd get data of type `TEXT` in THINK C:

```
Handle myData;
…
gClipboard->GetData('TEXT', &myData);
```

ConvertGlobal

```
procedure ConvertGlobal;
void ConvertGlobal (void);
```

Convert data in the desk scrap and put it in the private scrap. The default method does nothing. If your application supports a private scrap, you must override this method.

ConvertPrivate

```
procedure ConvertPrivate;
void ConvertPrivate (void);
```

Convert data in the private scrap and put it into the desk scrap. The default method does nothing. If your application supports a private scrap, you must override this method.

EmptyGlobalScrap

```
procedure EmptyGlobalScrap;
void EmptyGlobalScrap (void)
```

Clear the contents of the desk scrap.

Variables

This instance variable has the same value as `hItems` but a different type. It's kept for compatibility with the previous version of the THINK Class Library.

Variable	Type	Description
`items`	`LongHandle`	Handle to the items in the cluster.

Methods

In addition to the insertion, deletion, and searching methods, CCluster implements iteration methods that let you apply a function to all the objects in a cluster.

Construction and destruction

ICluster

```
procedure ICluster;
void ICluster (void);
```

Initialize the cluster.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of a cluster, but not the items in it.

DisposeAll

```
procedure DisposeAll;
void DisposeAll (void);
```

Dispose of a cluster and all the items in it. This method sends a `Free` (in THINK Pascal) or `Dispose` (in THINK C) message to every item in the cluster.

DisposeItems

```
procedure DisposeItems;
void DisposeItems (void);
```

Dispose of the items in a cluster, but not the cluster.

Insertion and deletion

These methods add objects to and remove objects from the cluster. Remember that the items in the cluster are not in any particular order.

Add

```
procedure Add (theObject: CObject);
void Add (CObject *theObject);
```

Add `theObject` to the cluster.

Remove

```
procedure Remove (theObject: CObject);
void Remove (CObject *theObject);
```

Remove `theObject` from the collection if it is already in the cluster.

Membership

These methods let you determine whether a particular item is in the cluster.

Includes

```
function Includes (theObject: CObject): Boolean;
Boolean Includes (CObject *theObject);
```

Return `TRUE` if `theObject` is in the cluster.

FindItem

```
function FindItem (function MyFunc(obj: CObject):
   Boolean): CObject;
CObject* FindItem (TestFunc MyFunc);
```

Look for an item that satisfies a function. `FindItem` applies the function `MyFunc` to each item in the list and returns the first one that causes the function to return `TRUE`.

In THINK C, you must declare `MyFunc` like this:

```
Boolean MyBoolFunc (CObject *obj);
```

FindItem1

```
function FindItem1 (function MyFunc1(obj: CObject;
   theParam: Ptr); param: Ptr): CObject;
CObject* FindItem1 (TestFunc1 MyFunc1, long param);
```

Look for an item that satisfies a function. `FindItem1` applies the function `MyFunc1` to each item in the list and returns the first one that causes the function to return `TRUE`. `Param` is an extra parameter you can pass to `FindItem1` if `MyFunc` needs more information.

In THINK C, you must declare `MyFunc` like this:

```
Boolean MyFunc1 (CObject *obj, long param);
```

Offset

```
function Offset (theObject: CObject): longint;
long Offset (CObject *theObject);
```

Returns the index of `theObject` in the collection. If `theObject` is not in the collection, `Offset` returns `BAD_INDEX`. This is an internal method. In most cases you won't need to use this method.

Iteration

These methods let you apply a function to all of the objects in a cluster. The function you apply to the objects in the cluster should not affect the cluster itself.

To apply a message to all the items in a cluster, you need to write a message-sending function. It's up to you to make sure that all the items in the cluster can respond to the message.

For example, this is how you send a `Draw` message to all the items in a cluster, in THINK Pascal:

```
procedure Perform_Draw (thePane: CPane;
   area: Ptr)
begin
   thePane.Draw(RecPtr(area)^);
end

theCluster.DoForEach1 (Perform_Draw, @area);
```

Note

`RectPtr` is defined in `TCL.p` as `^Rect`.

And this is how you send a `Draw` message to all the items in a cluster, in THINK C:

```
void Perform_draw(CPane *thePane, Rect *area)
{
   thePane->Draw(area);
}

myCluster->DoForEach1(Perform_Draw,
    (long)(&myRect));
```

DoForEach

```
procedure DoForEach (procedure Proc(theObject:
   CObject));
void DoForEach (EachFunc Proc);
```

Apply the procedure `Proc` to each object in the collection. In THINK C, you must declare `MyFunc` like this:

```
void MyFunc(CObject *theObject);
```


DoForEach1

```
procedure DoForEach1 (procedure Proc(theObject:
   CObject; theParam: Ptr); param: Ptr);
void DoForEach1 (EachFunc1 Proc, long param);
```

Apply the procedure `Proc` to each object in the collection along with the parameter `param`. In THINK C, you must declare `MyFunc` like this:

```
void MyFunc1(CObject *theObject, long param);
```

CCollaborator 23

Introduction

CCollaborator is an abstract class that implements objects that depend on one another. This class provides a mechanism for an object to announce changes to other objects.

Heritage

Superclass	CObject
Subclasses	CBureaucrat
	CCollection

Using CCollaborator

This class implements a mechanism that lets an object announce changes to other objects. The object that other objects depend on is called the **provider**. The objects that depend on the provider are called **dependents**.

For example, suppose that your program displays the contents of a file as different kinds of graphs in different windows. Whenever the data in the file change, the windows that display the data should also change. The data file is the provider. The graph display windows are the dependents.

To specify a dependency, send the dependent object a `DependUpon` message with the provider as the argument. When the provider changes, send the provider a `BroadcastChange` message. That method sends each of the dependents a `ProviderChanged` message.

The `BroadcastChange` method takes a parameter called `reason` that lets the dependents know why they're being notified. Reasons are implemented as long integers. In C, you can use `#define` directives or enums. In Pascal, you can use `const` declarations.

Reason names follow two conventions. First, reason names begin with the name of their class, without the C. For example, all of CArray's reasons begin

with `array`. Second, each collaborator subclass has a reason named *classname*`LastChange`. For example, CArray defines a reason named `arrayLastChange`. Using `LastChange` ensures that your class and all its subclasses have unique reason numbers. Set `LastChange` to the last reason your class defines. When you define a subclass, set the subclass's first reason to the superclass's `LastChange` plus one. For example, these are the reason definitions for the CArray class in THINK C:

```
enum
{
   arrayInsertElement = 1,
   arrayDeleteElement,
   arrayMoveElement,
   arrayElementChanged,

   arrayLastChange = arrayElementChanged
};
```

Note that `arrayLastChange` is set to `arrayElementChanged`, the last reason.

These are the reasons for CRunArray, a subclass of CArray:

```
enum
{
   runArraySizeChanged = arrayLastChange + 1,

   runArrayLastChange = runArraySizeChanged
};
```

Note that CArray's first (and only) reason is `arrayLastChange + 1`.

The examples below illustrate how you might implement the graphing program described above. When the data in the data file change, the data file's `UpdateData` method sends a `BroadcastChange` message with the reason `dataUpdated`. The `BroadcastChange` method sends each of the data file's graphs, the data file's dependents, a `ProviderChanged` method, which updates the picture of the data in the graph.

This is what it might look like in THINK C:

```
enum
{  myFileUpdated = 1,
   myFilePointAdded,
   myFilePointDeleted,

   myFileLastChange = myFileDeleted
};
. . .

void CMyFile::UpdateData (MyDataPtr oldData,
   MyDataPtr newData)
{
   . . .
   BroadcastChange(myFileUpdated,
                   (void *) newData );
}

void CMyGraphPane::ProviderChanged (
   CCollaborator *aProvider,
   long reason, void *info)
{
   if (aProvider == itsDataFile)
      switch (reason)
      {
         case myFileUpdated:
            UpdateGraphData(info);
            break;
         case myFilePointAdded:
            AddGraphData(info);
            break;
         case myFilePointDeleted:
            DeleteGraphData(info);
            break;
      }
}
```

Be sure to check the class of the provider. You can make sure that the provider is a known object as in the example here, or you can check for membership with the `member()`*. The preferred way is to check that it is a known object.*

This is the same example in THINK Pascal:

```
const
   myFileUpdated = 1;
   myFilePointAdded = myFileUpdated + 1;
   myFilePointDeleted = myFilePointAdded + 1;

   myFileLastChange = myFileDeleted;
. . .
```

```
procedure CMyFile.UpdateData (oldData,
   newData: MyDataPtr)
begin
   . . .
   BroadcastChange(longint(myFileUpdated),
                   Ptr (newData) );
end;

procedure CMyGraphPane.ProviderChanged (
   aProvider: CCollaborator;
   reason: longint; info: Ptr)
begin
   if aProvider = itsDataFile then
      case MyReason (reason) of
         myFileUpdated:
            UpdateGraphData(info);
         myFilePointAdded:
            AddGraphData(info);
         myFilePointDeleted:
            DeleteGraphData(info);
      end
end;
```

Be sure to check the class of the provider. You can make sure that the provider is a known object as in the example here, or you can check for membership with the `member()`. *The preferred way is to check that it is a known object.*

Variables

Variable	Type	Description
`itsProviders`	`CList`	The objects this collaborator depends upon.
`itsDependents`	`CList`	The objects that depend upon this collaborator.

Methods

Creation and destruction

ICollaborator

```
procedure ICollaborator;
void ICollaborator (void);
```

Initialize a collaborator. This method sets both `itsProviders` and `itsDependents` to NIL.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Remove this object from each of its provider's list of dependents and from each of its dependent's list of providers. Then dispose of its own lists of providers and dependents.

Creating dependency method

DependUpon

```
procedure DependUpon (aProvider: CCollaborator);
void DependUpon (CCollaborator *aProvider);
```

This collaborator depends on `aProvider`. Add this collaborator to `aProvider`'s list of dependents, and add `aProvider` to this collaborator's list of providers. If either of those lists are NIL, this method creates them before adding to them.

CancelDependency

```
procedure CancelDependency (aProvider: CCollaborator);
void CancelDependency (CCollaborator *aProvidoer);
```

This collaborator no longer depends upon `aProvider`. Remove this collaborator from the provider's dependent list and remove the provider from this collaborator's provider list.

Change notification methods

BroadcastChange

```
procedure BroadcastChange (reason: long; info: Ptr);
void BroadcastChange (long reason, void* info);
```

Notify this object's dependents that this object has changed. `Reason` is an integer that describes the type of change.You must define reasons for your subclass. `Info` is a pointer to any additional information needed to respond to the change.

If you want objects that aren't in a collaborator's list of dependents to know about a change, override this method. For example, CBureaucrat overrides this method to notify the bureaucrat's supervisor, in addition to its dependents, of the change.

ProviderChanged

```
procedure ProviderChanged (aProvider: CCollaborator;
   reason: longint; info: Ptr);
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

One of this object's providers has just changed. Your subclass must override this method to respond to each reason it can handle. The default method does nothing. `AProvider` is the provider that changed. `Reason` is an integer that describes the type of change. You must define reasons for your subclass. `Info` is a pointer to any additional information that your subclass might need.

List update methods

CCollaborator uses these methods internally to maintain the lists of dependents and providers. Whenever possible, use the `DependUpon` method instead of these methods. When a dependent is added or removed,

DependUpon and update *both* the dependent's list of providers *and* the provider's list of dependents. They do the same when a provider is added or removed. If you use the functions below, you might damage the collaborators' lists so that they don't match.

AddDependent

```
procedure AddDependent (aDependent: CCollaborator);
void AddDependent (CCollaborator *aDependent);
```

Add aDependent to this collaborator's list of dependents.

RemoveDependent

```
procedure RemoveDependent (aDependent: CCollaborator);
void RemoveDependent (CCollaborator *aDependent);
```

Remove aDependent from this collaborator's list of dependents.

AddProvider

```
procedure AddProvider (aProvider: CCollaborator);
void AddProvider (CCollaborator *aProvider);
```

Add aProvider to this collaborator's list of providers.

RemoveProvider

```
procedure RemoveProvider (aProvider: CCollaborator);
void RemoveProvider (CCollaborator *aProvider);
```

Remove aProvider from this collaborator's list of providers.

CCollection 24

Introduction

CCollection is an abstract class for implementing collections of things.

Heritage

Superclass	CCollaborator
Subclasses	CArray

Using CCollection

CCollection is an abstract class that helps you implement collections of things. The class doesn't provide the implementation of the collection. In your subclass, you specify the data structures that implement the list.

Variables

The only instance variable in this abstract class is the number of items in the collection. Your subclass will need to add at least the instance variable that contains or points to the memory that contains the items in your collection.

Variable	Type	Description
numItems	longint	The number of items in the collection.

Methods

ICollection

```
procedure ICollection;
void ICollection (void);
```

Initialize the collection. The default method sets the initial number of items to 0.

GetNumItems

```
function GetNumItems: longint;
long GetNumItems (void);
```

Return the number of items in a collection

IsEmpty

```
function IsEmpty: Boolean;
Boolean IsEmpty (void);
```

Return TRUE if the collection has no items.

CControl 25

Introduction

CControl is an abstract class for implementing Macintosh controls. The two built-in descendant classes, CButton and CScrollBar, implement the standard Macintosh controls.

Heritage

Superclass	CPane
Subclasses	CButton
	CScrollBar

Using CControl

This class describes the common methods for all controls. Scroll bars, buttons, check boxes, and radio buttons are descendants of CControl. Although you won't be creating subclasses of CControl (unless you define your own kind of control) you should become familiar with these methods.

Variables

The only instance variable for this class is a handle to the actual Macintosh control.

Variable	Type	Description
`macControl`	`ControlHandle`	Handle to the Macintosh control.

Methods

You can use these methods with any kind of control, but some methods don't really apply to all controls. For example, it's possible to set the value of a push button or to give a title to a scroll bar, but in most cases these actions don't make sense.

Construction and destruction methods

The CControl class does not define an initialization method. Each descendant of CControl defines an initialization method that uses CNTL resources to specify the type of control.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of a control.

Accessing methods

SetValue

```
procedure SetValue (aValue: integer);
void SetValue (short aValue);
```

Set the value of a control. This method sends a `BroadcastChange` message to all of this object's dependents with `controlValueChanged` as the reason.

GetValue

```
function GetValue: integer;
short GetValue (void);
```

Get the value of a control

SetMaxValue

```
procedure SetMaxValue (aMaxValue: integer);
void SetMaxValue (short aMaxValue);
```

Set the maximum value a control can have. For push buttons, check boxes, and radio buttons this value should be `1`.

GetMaxValue

```
function GetMaxValue: integer;
short GetMaxValue (void);
```

Get the maximum value a control can take on.

SetMinValue

```
procedure SetMinValue (aMinValue: integer);
void SetMinValue (short aMinValue);
```

Set the minimum value a control can have. Note that for push buttons, check boxes, and radio buttons this value should be `0`.

GetMinValue

```
function GetMinValue: integer;
short GetMinValue (void);
```

Get the minimum value a control can take on.

SetTitle

```
procedure SetTitle (aTitle: Str255);
void SetTitle (Str255 aTitle);
```

Set the title of a control. Scroll bars can have titles, but the title is not displayed.

GetTitle

```
procedure GetTitle (var aTitle: Str255);
void GetTitle (Str255 aTitle);
```

Get the title of a control

SetActionProc

```
procedure SetActionProc (anActionProc: ProcPtr);
void SetActionProc (VoidFunc anActionProc);
```

Set the action proc of a control. The action proc is called repeatedly while the mouse is held down in a control. The control manager distinguishes between an action proc called when the mouse goes down in a moving indicator (like a scroll box) and one called when the mouse goes down in a stationary part of a control. The default `DoClick` method sends the action proc to the control manager only for mouse hits that are not in a moving indicator. In THINK Pascal, you must declare the action proc like this:

```
procedure MyAction (macControl: ControlHandle;
   whichPart: integer);
```

In THINK C, you must declare the action proc like this:

```
pascal void MyAction (ControlHandle macControl,
   short whichPart);
```

If you want an action proc called when the mouse is in a moving indicator, you'll need to override the `DoClick` method.

Note

Be sure you understand action procs and the control manager before you use this method.

Appearance methods

These methods control the appearance of controls on the screen. They hide and show the control, draw it, move it, and change its size.

Hide

```
procedure Hide;
void Hide (void);
```

Hide the control.

Show

```
procedure Show;
void Show (void);
```

Show the control.

Activate

```
procedure Activate;
void Activate (void);
```

Make the control active. The control gets an `Activate` message automatically when its enclosure gets an `Activate` message.

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

Make the control inactive. The control gets a `Deactivate` message automatically when its enclosure gets a `Deactivate` message.

Offset

```
procedure Offset (hOffset, vOffset: longint;
   redraw: Boolean);
void Offset (long hOffset, long vOffset,
   Boolean redraw);
```

Move the control by `hOffset`, `vOffset` pixels. If `redraw` is `TRUE`, redraw the control after the move.

ChangeSize

```
procedure ChangeSize (delta: Rect; redraw: Boolean);
void ChangeSize (Rect *delta, Boolean redraw);
```

Change the size of a control. Each component of the rectangle specifies by how many pixels to offset each point. Positive numbers mean down and to the right. Negative numbers mean up and to the left. If `redraw` is `TRUE`, redraw the control after the change.

Draw

```
procedure Draw (var area: Rect);
void Draw (Rect *area);
```

Draw the control. The `area` parameter is ignored.

DrawAll

```
procedure DrawAll (var area: Rect);
void DrawAll (Rect *area);
```

Draw the control and all its subviews.

Prepare

```
procedure Prepare;
void Prepare (void);
```

Prepare the port and the coordinate system before drawing. Generally, you don't need to use or override this method.

Click response methods

The CControl class takes care of all the mouse tracking within a control.

DoClick

```
procedure DoClick (hitPt: Point;
   modifierKeys: integer; when: longint);
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handle a click in a control. `HitPt` is the point in frame coordinates. `ModifierKeys` is the same as the modifier field of an event record. `When` is the time that the mouse went down in ticks.

The default method handles both simple controls and controls with moving indicators and behaves a little differently in each case.

If the mouse goes down in a part other than a moving indicator:

- The default method calls the Toolbox routine `TrackControl` with the action proc you specified with the `SetActionProc` method.
- If `TrackControl` returns `TRUE` (the user clicks and releases the mouse in the same part of the control), this method sends a `DoGoodClick` message to the control.

If the mouse goes down in a moving indicator, like the thumb of a scroll bar:

- The default method calls the Toolbox routine `TrackControl` with no action proc.
- If the value of the control has changed (the user moved the indicator to a new place), this method sends a `DoThumbDragged` message to the control.

DoThumbDragged

```
procedure DoThumbDragged (delta: integer);
void DoThumbDragged (short delta);
```

An indicator in a control has been moved. The `DoClick` method sends a `DoThumbDragged` message to the control when the user changes the position of a control's indicator.

The default method does nothing. Controls with indicators should override this method. For an example of how to override this method, see `CScrollBar.DoThumbDragged` on page 411.

DoGoodClick

```
procedure DoGoodClick (whichPart: integer);
void DoGoodClick (short whichPart);
```

The mouse went down and up in the same part of a control.

The default method does nothing. Subclasses should override this method. See the `CButton.DoGoodClick` on page 221 for an example.

CDataFile 26

Introduction

This class implements the methods you need to work with a Macintosh data file. You can use this class without creating a subclass if you need to read raw bytes. If you require a structured file, you should create a subclass of CDataFile.

Heritage

Superclass	CFile
Subclasses	CPNTGFile
	CPictFile

Using CDataFile

You can use this class without creating a subclass to read and write the data fork of a Macintosh file. After creating an instance of CDataFile, use one of the specification methods inherited from CFile to indicate which file you're working with. Then use the Open method to open the file, and use the reading and writing methods to read and write the file.

Some CDataFile methods use the THINK Class Library's exception handling library if something goes wrong in a file operation. For more information on exception handling, see Chapter 8, "Exception Handling."

Note

This version of CDataFile handles errors differently than the version included with THINK C 4.0. The old version is now named ODataFile and is in the `Compatibility` foler.

Variables

The only instance variable for this class is the `refNum` of the file. The Macintosh file system uses this number to identify the volume, directory, and file

on disk. To learn more about the Macintosh file system, see *Inside Macintosh IV*, Chapter 19, "The File Manager."

Variable	Type	Description
`refNum`	`integer`	The file's `refNum` when open.

Methods

These methods let you open and close data files, read and write data, and get information about a file. You need to use the methods inherited from CFile to specify which file you want to work with.

Construction and destruction methods

IDataFile

```
procedure  IDataFile;
void IDataFile (void);
```

Initialize the data file.

Accessing methods

SetLength

```
procedure SetLength (aLength: longint);
void SetLength (long aLength);
```

Set the end-of-file marker for this file at `aLength`.

GetLength

```
function GetLength: longint;
long GetLength (void);
```

Return the length of the file. The file must be open already.

SetMark

```
procedure SetMark (howFar: longint;
   fromWhere: integer);
void SetMark (long howFar, short fromWhere);
```

Specify where subsequent read/write operations will take place. This position is called "the mark."

`HowFar` specifies how far in bytes from the `fromWhere` parameter to set the mark. Positive values are offsets toward the end of the file. Negative values are offsets toward the beginning of the file. `FromWhere` is one of the following:

fromWhere value	Meaning
`fsFromStart`	from the beginning of the file
`fsFromLEOF`	from the end of the file
`fsFromMark`	from the current position

GetMark

```
function GetMark: longint;
long GetMark (void);
```

Return the current position of the mark in `markPos`.

Open and close methods

Open

```
procedure Open (permission: SignedByte);
void Open (SignedByte permission);
```

Open the file with the given permission. `Permission` can be one of:

Permission	Meaning
`fsCurPerm`	same as the current permission
`fsRdPerm`	read permission
`fsWrPerm`	write permission
`fsRdWrPerm`	read/write permission
`fsRdWrShPerm`	shared read/write permission

See *Inside Macintosh IV*, Chapter 19, "The File Manager" for a discussion of permissions.

Close

```
procedure Close;
void Close (void);
```

Write out any unwritten data and close the file.

Read and write methods

ReadAll

```
function ReadAll: Handle;
Handle ReadAll (void);
```

Read the entire contents of the file into a handle. This method allocates the handle. You must pass in a reference to a handle. This example shows how to use `ReadAll` in THINK Pascal:

```
var
   theData: Handle;
begin
   …
   myDataFile.ReadAll(theData);
   …
end
```

And this example shows how to use `ReadAll` in THINK C:

```
Handle theData;
…
myDataFile->ReadAll(&theData);
```

ReadSome

```
procedure ReadSome (info: Ptr; var howMuch: longint);
void ReadSome (Ptr info, long howMuch);
```

Read howMuch bytes into a buffer starting at info. You must allocate the space to read the information into. This example shows how to use ReadSome in THINK Pascal::

```
var
   myBuffer = packed array [1..128] of char;
begin
   …
   myDataFile.ReadSome(@myBuffer, 128);
   …
end
```

And this example shows how to use ReadSome in THINK C:

```
char myBuffer[128];
…
myDataFile->ReadSome(myBuffer, 128L);
```

WriteAll

```
procedure WriteAll (contents: Handle);
void WriteAll (Handle contents);
```

Write the contents of the handle to the file.

WriteSome

```
procedure WriteSome (info: Ptr; var howMuch: longint);
void WriteSome (Ptr info, long howMuch);
```

Write howMuch bytes from the buffer starting at info.

CDecorator 27

Introduction

CDecorator implements an object that arranges windows on the screen. There is only one instance of this class.

Heritage

Superclass	CObject
Subclasses	None

Using CDecorator

CDecorator gives you methods for arranging windows on the screen. There is only one instance of CDecorator which is stored in the global variable gDecorator.

After creating and initializing a new window, send it in a PlaceNewWindow message to the decorator. The decorator makes the window a bit smaller than the main screen and offsets it down and to the right. Using the decorator is entirely optional.

You can create a subclass of CDecorator if you like. For instance, you might want a decorator that implements window tiling. Be sure to override the MakeDecorator method in your application subclass to initialize your decorator and to store the decorator in the global variable gDecorator.

Variables

The global decorator object, created in the application method MakeDecorator, is stored in the global variable gDecorator.

Global variable

Variable	Type	Description
gDecorator	CDecorator	The window-dressing object.

Instance variables

Variable	Type	Description
wCount	integer	The number of new windows placed
index	integer	Index for offsetting windows
wWidth	integer	Width of a window in pixels
wHeight	integer	Height of a window in pixels
hLocation	integer	Horizontal location for next window
vLocation	integer	Vertical location for next window

Methods

IDecorator

```
procedure IDecorator;
void IDecorator (void);
```

Initialize the decorator. The application method `MakeDecorator` creates the global decorator and sends it this message.

PlaceNewWindow

```
procedure PlaceNewWindow (theWindow: CWindow);
void PlaceNewWindow (CWindow *theWindow);
```

Place `theWindow` on the screen. The default method makes it fill up most of the screen and calls `StaggerWindow` to position it. .

StaggerWindow

```
procedure StaggerWindow (theWindow: CWindow);
void StaggerWindow (CWindow *theWindow);
```

Position a new window on the screen, offset down and to the right of the last window the decorator placed.

CenterWindow

```
procedure CenterWindow (theWindow: CWindow );
void CenterWindow (CWindow *theWindow);
```

Center the window specified by `theWindow` on the main screen. If `theWindow` is a modal dialog, this method tries to put it in the top third of the main screen. This method does not increment `wCount` or change any of the instance variables.

GetWCount

```
function GetWCount: integer;
short GetWCount (void);
```

Return the number of windows the decorator has put on the screen. You can use this value to give your untitled windows sequential numbers like "Untitled-1."

CDesktop 28

Introduction

CDesktop implements a view that occupies the entire screen. This view serves as the top of the visual hierarchy and the enclosure for all windows. Normally, there is only one instance of CDesktop.

Heritage

Superclass	CView
Subclasses	CFWDesktop

Using CDesktop

The most common way you'll use the desktop is as the enclosure for your windows. Your application should not send messages to the desktop. Instead, you should rely on higher level objects like windows and directors to send messages to it.

CFWDesktop is a subclass of CDesktop which implements a desktop that supports floating windows. CFWDesktop is described on page 315.

The desktop is stored in the global variable `gDesktop`. The CApplication method `MakeDesktop` creates an instance of the CDesktop and stores it in that variable.

Variables

The global variable `gDesktop` holds a pointer to the single instance of the desktop. Use this variable to specify the enclosure for your application's windows.

Global variable

Variable	Type	Desktop
gDesktop	CDesktop	The single instance of the desktop

Instance variables

Variable	Type	Desktop
bounds	Rect	Boundaries of the desktop
itsWindows	CList	List of windows
topWindow	CWindow	Top-most application window

Methods

Construction and destruction methods

IDesktop

```
procedure IDesktop (aSupervisor: CBureaucrat);
void IDesktop (CBureaucrat *aSupervisor);
```

Initialize the desktop. The default method opens a `GrafPort` that your application uses as its desktop. The desktop's supervisor should be the application, stored in the global variable `gApplication`.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the desktop and send this message to all of the desktop's windows.

Appearance methods

Show

```
procedure Show;
void Show (void);
```

Makes a desktop visible by showing all of its windows. This method sends a `Show` message to all of the desktop's windows.

Hide

```
procedure Hide;
void Hide (void);
```

Hide a desktop by hiding all of its windows. This method sends a `Hide` message to all of the desktop's windows.

Activate

```
procedure Activate;
void Activate (void);
```

Activate the desktop by activating the top window. This method sends an `Activate` message to the desktop's top window.

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

Deactivate a desktop by deactivating its top window. This method sends a `Deactivate` message to the desktop's top window.

ReallyVisible

```
function ReallyVisible: Boolean;
Boolean ReallyVisible (void);
```

Return TRUE if the desktop is visible.

Mouse methods

DispatchClick

```
procedure DispatchClick (var macEvent: EventRecord);
void DispatchClick (EventRecord *macEvent);
```

If the user clicks the mouse anywhere, find out where the mouse went down, and send the appropriate message to the object the click is intended for. If the top window is modal, and the mouse did not go down in the menu bar, this method beeps and does not process the click.

This method uses what the Macintosh Toolbox routine `FindWindow` returns to determine what message to send to which object.

Part	Action
inDesk	If the mouse goes down in the desktop, this method sends a `DoClick` message to the desktop. The default `DoClick` method does nothing.
inSysWindow	If the mouse goes down in a system window, this method calls the Toolbox routine `SystemClick`.
inMenuBar	If the mouse goes down in the menu bar, this method calls the Toolbox routine `MenuSelect`. This method then sends a `FindCmdNumber` message to the bartender to convert the menu selection into a command number. This command number is sent to the gopher in a `DoCommand` message.
inContent	If the mouse goes down in the content region of a window and the window is not active, this method sends the window a `Select` message. If the window is already active and can receive clicks, this method sends the window a `DispatchClick` message which will

eventually send a `DoClick` message to a pane or the window itself.

`inDrag`	If the mouse goes down in the drag region (the title bar), this method sends the window a `Drag` message.
`inGrow`	If the mouse goes down in the window's grow region (the lower right corner), this method sends the window a `Resize` message.
`inGoAway`	If the mouse goes down and up in the window's close box, this method sends it a `Close` message.
`inZoomIn, inZoomOut`	If the mouse goes down and up in the window's zoom box, this method sends it a `Zoom` message.

DoMouseUp

```
procedure DoMouseUp (macEvent: EventRecord);
void DoMouseUp (EventRecord *macEvent);
```

Handle a mouse up event in the desktop. The default method does nothing.

DispatchCursor

```
procedure DispatchCursor (where: Point;
   mouseRgn: RgnHandle);
void DispatchCursor (Point where, RgnHandle mouseRgn);
```

Set the cursor for the view the cursor is in. If the cursor is over an active window, this method sends a `DispatchCursor` message to the window. If the cursor is in the menu bar, this method sets it to the arrow. If the cursor is not in the menu bar or in an active window, this method sends an `AdjustCursor` message to the desktop. You should not override this method.

AdjustCursor

```
procedure AdjustCursor (where: Point;
   mouseRgn: RgnHandle);
void AdjustCursor (Point where, RgnHandle mouseRgn);
```

Adjust the cursor shape when the mouse is in an insignificant part of the desktop. The default method sets the cursor to an arrow.

Contains

```
function Contains (thePoint: Point): Boolean;
Boolean Contains (Point thePoint);
```

Return `TRUE` if `thePoint` is within the bounds of the desktop.

HitSamePart

```
function HitSamePart (var pointA, pointB: Point):
   Boolean;
Boolean HitSamePart (Point pointA, Point pointB);
```

Return `TRUE` if `pointA` and `pointB` are reasonably close together. "Reasonably close" means that the two points are within `REASONABLY_CLOSE` pixels.

Window methods

You do not have to use any of these methods yourself. Directors, objects that manage the interaction between windows and the desktop, take care of sending most of these messages.

AddWind

```
procedure AddWind (theWindow: CWindow);
void AddWind (CWindow *theWindow);
```

Add `theWindow` to the desktop's window list.

RemoveWind

```
procedure RemoveWind (theWindow: CWindow);
void RemoveWind (CWindow *theWindow);
```

Remove `theWindow` from the desktop's window list.

SelectWind

```
procedure SelectWind (theWindow: CWindow);
void SelectWind (CWindow *theWindow);
```

Select `theWindow` and bring `theWindow` to the front.

ShowWind

```
procedure ShowWind (theWindow: CWindow);
void ShowWind (CWindow *theWindow);
```

Make `theWindow` visible on the desktop.

HideWind

```
procedure HideWind (theWindow: CWindow);
void HideWind (CWindow *theWindow);
```

Hide `theWindow`.

DragWind

```
procedure DragWind (theWindow: CWindow;
   macEvent: EventRecord);
void DragWind (CWindow *theWindow,
   EventRecord *macEvent);
```

Drag `theWindow` on the desktop.

UpdateWindows

```
procedure UpdateWindows;
void UpdateWindows (void);
```

Send an `Update` message to each of the desktop's windows.

Accessing methods

GetTopWindow

```
function GetTopWindow: CWindow;
CWindow* GetTopWindow (void);
```

Get the frontmost window.

GetBounds

```
procedure GetBounds (theBounds: Rect);
void GetBounds (Rect *theBounds);
```

Get the bounds of the desktop.

GetAperture

```
procedure GetAperture (var theAperture: LongRect);
void GetAperture (LongRect *theAperture);
```

Get the visible portion of the desktop. This method returns the bounds of the desktop since all of the desktop is always visible.

Calibration methods

Prepare

```
procedure Prepare;
void Prepare (void);
```

Set the port to the desktop port. You should not use or override this method.

Cleanup method

Cleanup

```
procedure Cleanup;
```

This method is in THINK Pascal only. This method does nothing, but CFWDesktop overrides it to move THINK Pascal windows to the back. For more information see `CFWDesktop.Cleanup` on page 318. THINK C doesn't use this method since CFWDesktop doesn't need to move THINK C windows to the back.

CDirector 29

Introduction

CDirector is an abstract class for implementing a window that can handle commands. Directors implement communication between the application and a window.

Heritage

Superclass	CDirectorOwner
Subclasses	CDocument
	CClipboard
	CTearOffMenu

Using CDirector

A director is an abstract class that manages the communication between the application and a window. Any time you want to display a window, it must be related to a director.

In most cases, you'll use the CDocument subclass to display and manipulate data stored in a file in a window. The only time you'll create a subclass of CDirector is when you need a special kind of window like a status window, a subwindow, or a tear-off menu.

Note

The class CTearOffMenu implements a tear-off menu as a subclass of CDirector.

When a window belonging to a director becomes the active window, the gopher points to the bureaucrat specified by the director's `itsGopher` instance variable. When the window becomes inactive, the gopher points to the application.

When the switchboard processes a window-related event (such as an update event), it sends the message to the window which in turn sends the message

to the director. When the user chooses a command from the menu, the switchboard sends the DoCommand message to the gopher.

The supervisor of a director must be the application. The supervisor of a window must be a subclass of CDirector.

Variables

Global variable

The global variable gGopher points to the current bureaucrat which is usually a descendant of CDirector. When a window becomes active, gGopher points to the bureaucrat specified in the director's itsGopher instance variable. When there are no active directors, gGopher points to the application.

Variable	Type	Description
gGopher	CBureaucrat	The current bureaucrat.

Instance variables

The instance variable itsGopher usually points to the main pane of a window. When the director becomes active, the bureaucrat specified in itsGopher becomes the gopher.

Variable	Type	Description
itsWindow	CWindow	Window that the director controls
active	Boolean	TRUE if the director is active
itsGopher	CBureaucrat	Bureaucrat to make the gopher when the director is activated.
activateWindOnResume	Boolean	TRUE, if the director activates its window when the program is resumed

Methods

Creation and destruction

IDirector

```
procedure IDirector (aSupervisor: CDirectorOwner);
void IDirector (CDirectorOwner *aSupervisor);
```

Initialize the director. This method adds the director to aSupervisor's list of directors. By default, the director has no window and is not active. The itsGopher instance variable is set to this object. Your director subclass must create the window for the director. Your director subclass should call the inherited method to initialize the director.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the director. This method sends a Free or Dispose message to the director's window (if it has one) and removes the director from the supervisor's list of directors. Your subclass must call the inherited method to make sure that the director is disposed of properly.

Accessing method

GetWindow

```
function GetWindow: CWindow;
CWindow* GetWindow (void);
```

Return the window that the director controls.

FindViewByID

```
function FindViewByID (aViewID: longint): CView;
CView* FindViewByID (long aViewID);
```

Within the director's window, locate the view with id aViewID.

OwnsWindow

```
function OwnsWindow (aWindow: CWindow): Boolean;
Boolean OwnsWindow (CWindow *aWindow);
```

Return TRUE if this director owns aWindow. The director owns the window if the window's supervisor is this object.

Command methods

DoCommand

```
procedure DoCommand (theCommand: longint);
void DoCommand (long theCommand);
```

Handle a command. The default method handles this command:

Command	Description
cmdClose	Send a Close(FALSE) message to the director

Your director class will usually override this method. You should handle your own commands first, then call the inherited method to get the generic effects.

UpdateMenus

```
procedure UpdateMenus;
void UpdateMenus (void);
```

If a window is associated with the director, this method enables the **Close** command (`cmdClose`).

Appearance methods

Activate

```
procedure Activate;
void Activate (void);
```

A director is becoming active. If the director's window is not a floating window, sends a `BecomeGopher(TRUE)` message to `itsGopher`. This method sets `gSleepTime` to 0 to force an idle event and sends an `ActivateDirector` message to the director's supervisor.

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

A director is becoming inactive. If the director is the gopher, send a `BecomeGopher(TRUE)` message to the director's supervisor. This method sends as `DeactivateDirector` message to the director's supervisor.

ActivateDirector

```
procedure ActivateDirector (aDirector: CDirector);
void ActivateDirector (CDirector *aDirector);
```

A director owned by this director is becoming active. This method calls the inherited method to make `aDirector` the frontmost director and sends an `ActivateDirector` message to this director's supervisor.

DeactivateDirector

```
procedure DeactivateDirector (aDirector: CDirector);
void DeactivateDirector (CDirector *aDirector);
```

A director owned by this director is becoming inactive. This method calls the inherited method and sends a `DeactivateDirector` message to this director's supervisor.

Suspend

```
procedure Suspend;
void Suspend (void);
```

The application is being suspended. This method calls the inherited method to send a `Suspend` message to all of the supervisor's directors. If this director is active, and if it owns an active window, it sends the window a `Deactivate` message. The director remains active even though the application is suspended.

Resume

```
procedure Resume;
void Resume (void);
```

The application is being resumed. This method calls the inherited method to send a `Resume` method to all of the supervisor's directors. If the director is active and the director owns a window, send the window an `Activate` message.

The default method sends the director an `Activate` message.

Close

```
function Close (quitting: Boolean): Boolean;
Boolean Close (Boolean quitting);
```

Close a director. This method calls the inherited method to close all of the directors that the supervisor owns (including this one). If all of the directors closed, this method calls `Free` or `Dispose` and returns TRUE, otherwise it returns FALSE.

Window methods

CloseWind

```
procedure CloseWind (theWindow: CWindow);
void CloseWind (CWindow *theWindow);
```

This method calls `Close(FALSE)`. If `theWindow` is not the window owned by this director, it sends a Dispose or a Free message to `theWindow`.

If you want a click in the window's close box to mean something other than closing the director, override this method. For instance, you might want to override this method so it hides the window instead of closing it. See CClipboard's `CloseWind` method on page 238 for an example.

ActivateWind

```
procedure ActivateWind (theWindow: CWindow);
void ActivateWind (CWindow *theWindow);
```

The window owned by the director has been activated. You generally won't need to use or override this method in your director subclasses unless you want to do something to `theWindow` (and not the director) when it becomes active. To perform specific actions at activate time, override the `Activate` method instead.

DeactivateWind

```
procedure DeactivateWind (theWindow: CWindow);
void DeactivateWind (CWindow *theWindow);
```

The window owned by the director is becoming inactive. You generally won't need to use or override this method in your director subclasses unless you want to do something to `theWindow` (and not the director) when it becomes deactivated. To perform some specific actions at deactivate time, override the `Deactivate` method instead.

IsActive

```
function IsActive: Boolean;
Boolean IsActive (void);
```

Returns TRUE if the director is active; FALSE otherwise. You should not override this method.

Change notification method

This method overrides a method in CCollaborator. For more information, see CCollaborator on page 249.

ProviderChanged

```
procedure ProviderChanged (aProvider: CCollaborator;
   reason: longint; info: Ptr);
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

One of this director's providers or subordinates has just changed. This method handles the reason `bureaucratIsGopher` by updating the instance variable `itsGopher`. If your subclass handles other reasons, it should override this method and call the inherited method.

`AProvider` is the provider or subordinate that changed. In this case it is the bureaucrat that is becoming the gopher `Reason` is an integer that describes the type of change. In this case, `bureaucratIsGopher` is the only reason handled. `Info` is a pointer to any additional information that a subclass might need. This method does not use `info`, and passes it along in a call to the inherited method.

CDirectorOwner 30

Introduction

CDirectorOwner is an abstract class for objects that own directors, which are objects that manage windows.

Heritage

Superclass	CBureaucrat
Subclasses	CApplication
	CDirector

Using CDirectorOwner

CDirectorOwner is an abstract class for objects that own directors. A director is an object that manages the communication between an application and a window.

CDirectorOwner has two subclasses: CApplication and CDirector. An application needs to inform its directors when the application is suspending, resuming, or quitting. If your application implements multi-window documents, use an object of class CDocument as the main document and objects of class CDirector as the subwindows. The subwindows should be owned by the document object.

Variables

Variable	Type	Description
itsDirectors	CList	List of the directors this object owns.
active	Boolean	TRUE if any director is active.

Methods

Creation and destruction

IDirectorOwner

```
procedure IDirectorOwner (aSupervisor:
   CDirectorOwner);
void IDirectorOwner (CDirectorOwner *aSupervisor);
```

Initialize the owner. This methodsets `itsDirectors` to NULL.

Dispose/Free

```
procedure Free;
void Dispose (void);
```

Dispose of this object and all the directors it owns.

Insertion and deletion methods

AddDirector

```
procedure AddDirector (aDirector: CDirector);
void AddDirector (CDirector *aDirector);
```

Add a new director to the director list.

RemoveDirector

```
procedure RemoveDirector (aDirector: CDirector);
void RemoveDirector (CDirector *aDirector);
```

Remove a director from the director list.

Appearance methods

ActivateDirector

```
procedure ActivateDirector (aDirector: CDirector);
void ActivateDirector (CDirector *aDirector);
```

A director that this object owns has been activated. Sets `active` to TRUE.

DeactivateDirector

```
procedure DeactivateDirector (aDirector: CDirector);
void DeactivateDirector (CDirector *aDirector);
```

A director that this object owns has been deactivated. Sets `active` to FALSE.

Suspend

```
procedure Suspend;
void Suspend (void);
```

Notify this object's directors that the application is suspending. Send a `Suspend` message to all directors.

Resume

```
procedure Resume;
void Resume (void);
```

Notify this object's directors that the application is resuming. Send a `Resume` message to all the directors.

Quit

```
function Quit: Boolean;
Boolean Quit (void);
```

The application is about to quit. This method is the same as calling `Close(TRUE)`.

Close

```
function Close (fQuitting: Boolean): Boolean;
Boolean Close (Boolean fQuitting);
```

Try to close all the directors that this object owns. If they all close, this method returns TRUE. If any director doesn't close, this method stops closing directors and returns FALSE. Set `fQuitting` to TRUE if you are quitting the application.

CDocument 31

Introduction

CDocument is the main class for presenting and manipulating information. You can think of a document as the association of a window, a file, and a set of panes.

Your application must create a subclass of CDocument.

Heritage

Superclass	CDirector
Subclasses	You must create a subclass of CDocument

Using CDocument

CDocument is one of the classes you need to override to implement an application in the THINK Class Library. You can think of a document as a file that you view through a window. A better way to think about a document is that it is the essence of a Macintosh application. It is anything that you can display and manipulate inside a window.

The document is where your application draws and displays its data. Since documents are descendants of CDirector, all documents have windows associated with them. Most documents also have an associated file. Neither the window nor the file are created automatically. You must create them yourself in your document's initialization method.

Your document class should override these methods:

initialization method	`OpenFile`
`Free`	`DoSave`
`DoCommand`	`DoSaveAs`
`NewFile`	`Revert`

Your document class must have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By con-

vention, the name of your initialization method should be `I` *YourDoc*, where *YourDoc* is the name of your document class. Your initialization method should call `IDocument`. The supervisor of a document is always `gApplication`.

If your application allocates memory, you should also override the `Free` or `Dispose` method to deallocate it. Be sure that your method calls the inherited method to make sure that the document is disposed of properly.

Note

You do not need to dispose of the window or file associated with a document. The default `Free` or `Dispose` method does that for you.

Your document class's `DoCommand` method does most of the work in your application. When a window is active, the switchboard sends all commands to the document first (it's the gopher), and if the document can't handle it, the application class tries to handle it. Your document class should handle all the commands it knows about, and call the inherited method when it can't.

Note

Be sure that your `DoCommand` method or that one of the methods it invokes sets the instance variable `dirty` to `TRUE` when there has been a change to the document.

Your document class gets a `NewFile` message when you choose **New** from the **File** menu. This method needs to create a window and attach the panes for it. The `NewFile` method doesn't need to create a file until you try to save the document.

Your document gets an `OpenFile` message when you choose **Open...** from the **File** menu. The `OpenFile` method has one argument: a Macintosh `SFReply` record. When you get the `OpenFile` message, you can be sure that the `SFReply` record is properly filled in. Your `OpenFile` message needs to create an instance of a file object (usually of class CDataFile). You can send your file any of several read messages to get its contents. Your `OpenFile` method also needs to create a window to display the contents of the file, just as your `NewFile` method does.

When the you choose **Save** from the **File** menu, your document gets a `DoSave` message. Your `DoSave` method should write the contents of its file to disk. The file object is stored in the instance variable `itsFile`.

When the you choose **Save As...** from the **File** menu, your document gets a `DoSaveAs` message. This method takes an `SFReply` record, and you can be sure that it is properly filled in. Your document class needs to override this method to write its data to a file.

If your application supports the **Revert** command, you should implement `DoRevert` method. Your implementation might close the file without saving, then open the file again.

Variables

Global variables

The global variable `gGopher` points to the current bureaucrat which is usually a document. When a window becomes active, `gGopher` points to the document that owns it. When there are no active documents, `gGopher` points to the application.

The supervisor of every document is the application.

Variable	Type	Description
`gGopher`	`CBureaucrat`	The current bureaucrat.
`gApplication`	`CApplication`	The application

Instance variables

Variable	Type	Description
`itsMainPane`	`CPane`	The document's main pane. `NIL` if the document has no main pane. The main pane's enclosure should be `itsWindow`, an instance variable inherited from CDirector.
`itsFile`	`CFile`	The file associated with this document. `NIL` if document has no file.
`lastTask`	`CTask`	The last task this document was notified as being completed
`undone`	`Boolean`	`TRUE` if the last task was undone.
`itsPrinter`	`CPrinter`	The printer object associated with this document. `NIL` if document is not printable

Variable	Type	Description
dirty	Boolean	TRUE if document has been altered.
pageWidth	integer	The width of a page.
pageHeight	integer	The height of a page.

Methods

Construction and destruction methods

IDocument

```
procedure IDocument (aSupervisor: CApplication;
   printable: Boolean);
void IDocument (struct CApplication *aSupervisor,
   Boolean printable);
```

Initialize the document. `ASupervisor` must be the `gApplication`. If the value of `printable` is TRUE, this method calls `MakePrinter` to create an instance of CPrinter and stores it in the `itsPrinter` instance variable.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the document and memory it allocated. This method sends a `Free` or a `Dispose` message to `itsFile`, `lastTask`, and `itsPrinter` if they were allocated. The `Close` method usually calls this method. If your document subclass allocates memory (such as a pane for `itsMainPane`) your method must dispose of it, then you must call the inherited method to dispose of the rest of the document.

Command methods

Notify

```
procedure Notify (theTask: CTask);
void Notify (CTask *theTask);
```

A subordinate has completed a task. The default method disposes the current `lastTask` and stores `theTask` in the instance variable `lastTask`. This method sets undone to FALSE and dirty to TRUE.

DoCommand

```
procedure DoCommand (theCommand: longint);
void DoCommand (long theCommand);
```

Handle a document-related command. The default method handles these commands:

Command	Action
cmdClose	Send a `Close` message to the document.
cmdSave	Set the cursor to the watch cursor and send a `DoSave` message to the document.
cmdSaveAs	Send a `DoSaveFileAs` message to the document. The default `DoSaveFileAs` method sets the cursor to a watch.
cmdRevert	Display a "Do you really want to revert?" alert. If you respond OK, set the cursor to the watch cursor and send a `DoRevert` message to the document.
cmdPageSetup	If the document is printable, send a `DoPageSetup` to the document.
cmdPrint	If the document is printable, send a `DoPrint` message to the document.
cmdUndo	If there is a last task, send either a `Redo` or an `Undo` message to the task. Then send an `UpdateUndo` message to the document.

Your document class will usually override this method. You should handle your own commands first, then call the inherited method to handle the default commands.

UpdateMenus

```
procedure UpdateMenus;
void UpdateMenus (void);
```

Update the menu items right before they appear on the screen. The default method enables the following commands:

Command	Enabled if...
cmdSaveAs	always
cmdSave	the document is dirty.
cmdRevert	a file is associated with the document and it's dirty.
cmdPageSetup	the document is printable.
cmdPrint	the document is printable.
cmdUndo	there is a last task to undo.

Your document class should override this method to enable the appropriate commands for your document. Be sure you call the inherited method before you enable your document's commands.

Appearance Methods

Close

```
function Close (quitting: Boolean): Boolean;
Boolean Close (Boolean quitting);
```

A document is being closed. This method sends a `ConfirmClose` message to the document. If the result is `TRUE`, it sends a `Close` message to `itsFile` (if it has one). The `quitting` parameter tells the `ConfirmClose` method whether to ask to save before "closing" or "quitting." If the document was actually closed, this method returns TRUE. Otherwise it returns FALSE.

CloseWind

```
procedure CloseWind (theWindow: CWindow);
void CloseWind (CWindow *theWindow);
```

Close the specified window. The default method sends a `Close` message to the document. If you want to be able to close windows without actually closing a file, you should override this method. Otherwise, your document class should not override this method.

ConfirmClose

```
function ConfirmClose (quitting: Boolean): Boolean;
Boolean ConfirmClose (Boolean quitting);
```

Display a "Save before closing?" or "Save before quitting?" dialog. If you answer yes, send a `DoSave` to the document and return TRUE. If you answer no, just return TRUE. If you answer Cancel, return FALSE. If your document does not have a file, your document class should override this method with a method that always returns TRUE.

File Creation

NewFile

```
procedure NewFile;
void NewFile (void);
```

Open a new file. The default method does nothing. Your `CreateDocument` method in your application class should send a `NewFile` message to the document it creates. Your document class should override this method to do the following:

- Create a new window and assign it to `itsWindow`
- Create a new file and assign it to `itsFile`
- Create the panes you need and assign the main pane to `itsMainPane`

OpenFile

```
procedure OpenFile (macSFReply: SFReply);
void OpenFile (SFReply *macSFReply);
```

Open an existing file. The default method does nothing. Your `OpenDocument` method in your application class should send an `OpenFile` message to the document it creates. Your document class should override this method and do the following:

- Create a new window and assign it to `itsWindow`
- Open the file specified in `macSFReply` and assign it to `itsFile`
- Create the panes you need and assign the main pane to `itsMainPane`
- Display the contents of the file in the pane

Note

To learn how to specify a file, see the class CFile. To learn how to read and write from a data file, see the class CDataFile on page 263.

Printing Methods

MakePrinter

```
procedure MakePrinter;
void MakePrinter (void);
```

Make the printer object for this document. For more information on printer objects, see page 389. If you create subclass of CPrinter, you should override this method to create an object of your class.

Paginate

```
procedure Paginate;
void Paginate (void);
```

Send a `Paginate` message to `itsMainPane` if its not NIL, otherwise send a `ResetPagination` message to `itsPrinter`.

PageCount

```
function PageCount: integer;
short PageCount (void);
```

Return the number of pages in the document. This method sends a GetStripCount message to itsPrinter to find out how many pages the document will take. If there are too many pages (more than 999), it raises an exception.

If the document doesn't have a CPrinter object associated with it, this method returns zero.

AboutToPrint

```
procedure AboutToPrint (var firstPage, lastPage:
   integer);
void AboutToPrint (short *firstPage, short *lastPage);
```

Check the range of pages to be printed. The default method changes `lastPage` to be equal to `PageCount`. Your document class should override this method to do whatever is appropriate for your application. This is the place where the document can request information about the page size from `itsPrinter`.

PrintPageOfDoc

```
procedure PrintPageOfDoc (pageNum: integer);
void PrintPageOfDoc (short pageNum);
```

Print the specified page. The default method sends a `PrintPage` message to `itsMainPane` if it's not NIL.

DonePrinting

```
procedure DonePrinting;
void DonePrinting (void);
```

Printing is complete. The `PrintPageRange` method in CPrinter sends this message when the print loop is over. This method sends a `DonePrinting` message to `itsMainPane`.

Filing Methods

DoSave

```
function DoSave: Boolean;
Boolean DoSave (void);
```

Save the document under its current name and return `TRUE` if successful. The current file is available through the instance variable `itsFile`. The default method does nothing. Your document class must override this method.

DoSaveAs

```
function DoSaveAs (macSFReply: SFReply): Boolean;
Boolean DoSaveAs (SFReply *macSFReply);
```

Save the document under a new name and return `TRUE` if successful. The `macSFReply` record specifies where to write the file. The default method does nothing. Your document class must override this method.

DoRevert

```
procedure DoRevert;
void DoRevert (void);
```

Revert to the last saved version of this document. The default method does nothing. If you want your application to support the **Revert** command, you'll have to override this method.

DoSaveFileAs

```
function DoSaveFileAs: Boolean;
Boolean DoSaveFileAs (void);
```

Respond to a **Save As...** command and return `TRUE` if successful. This method sends a `PickFileName` message to the document. If you provide a good file name, it sends a `DoSaveAs` message to the document. Since this method implements the standard **Save As...** command through two other messages, you won't need to override this method.

PickFileName

```
procedure PickFileName (var macSFReply: SFReply);
void PickFileName (SFReply *macSFReply);
```

Display a standard get file dialog to get a new file name. `PickFileName` uses the `GetName` method to specify the default name.

GetName

```
procedure GetName (var theName: Str255);
void GetName (Str255 theName);
```

Get the name of the document. If there is a file associated with the document, this method returns the name of the file. If there is no file associated with the document, but there is a window associated with it, it returns the title of the window. If there is neither a file nor a window associated with the document, this method returns a null string.

Undo methods

UpdateUndo

```
procedure UpdateUndo;
void UpdateUndo (void);
```

Update the **Undo/Redo** menu item to have the correct wording. Default method sends a `GetNameIndex` message to the `lastTask` to find its string in the `STRtaskNames STR#` resource. Your document class should not override this method.

Class Resources

Resource	Description
`STRprompt 150`	`STR` resource ID for `PickFileName` prompt string
`ALRTrevert 150`	Revert to saved alert
`ALRTsaveChanges 151`	Save changes before close/quit alert
`STRtaskNames 130`	`STR#` resource ID for task names
`STRcommon 128`	`STR#` resource for commonly used strings

CEditText 32

Introduction

CEditText implements a pane that displays text. This class uses the Macintosh TextEdit routines.

Heritage

Superclass	CAbstractText
Subclasses	CStaticText

Using CEditText

Use CEditText whenever you need to display unformatted text. An edit text pane is usually the panorama in a scroll pane so you can scroll through the text. The `DoCommand` method of an edit text pane handles all the common text editing commands such as cutting and pasting, font selection, line spacing, etc. The `Specify` method, inherited from CAbstractText, on page 146, lets you choose whether the user can edit and copy your text pane's text.

CEditText does not use the Styled Text Edit routines described in *Inside Macintosh V*. To create a pane of styled text, use CStyleText.

To make sure that the edit text pane responds to commands, you must place it in the chain of command. The best way to do this is to set the value of your document's `itsGopher` instance variable to the edit pane.

Because CEditText is based on the Macintosh TextEdit (TE) routines, it has some limitations. It's designed to edit small amounts of text. The maximum number of characters you can store in a CEditText record is around 32,000, but you'll notice performance degradation long before it gets that big. To create a pane to display more text (for a text editor, for example), create a subclass of CAbstractText, described on page 143.

Variables

Variable	Type	Description
macTE	TEHandle	TextEdit record handle.
spacingCmd	longint	Line spacing command number.
alignCmd	longint	Alignment command number.

Methods

Construction and destruction methods

IEditText

```
procedure IEditText (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption;
   aLineWidth: integer);
```

```
void IEditText (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aLineWidth);
```

Initialize an edit text pane. Most of the arguments to this method are identical to the pane initialization. `ALineWidth` specifies how wide the lines should be. If it's less than zero, the width is the same as the Macintosh TE record's `viewRect`.

Note

The descriptions of the other arguments are in CPane on page 347.

IViewRes

```
procedure IViewRes (rType:ResType; resID: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
```

```
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize edit text pane from a resource template. `RType` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the enclosure and supervisor of the pane.

To initialize a text pane from a resource file, use a `'AbTx'` resource.

Note

Earlier versions of the THINK Class Library used `'StTx'` resources to initialize static text and edit text panes. You should make sure that any old programs do not use `'StTx'`.

MakeMacTE

```
procedure MakeMacTE;
```

```
void MakeMacTE (void);
```

Create a TextEdit record and set `macTE` to it.

Free/Dispose

```
procedure Free;
```

```
void Dispose (void);
```

Dispose the TextEdit record `macTE`.

Mouse and Keystrokes methods

DoClick

```
procedure DoClick (hitPt: Point;
   modifierKeys: integer; when: longint);
```

```
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handle a mouse down in an edit text pane. This method sends a `SelectionChanged` message after processing the click.

Command methods

PerformEditCommand

```
procedure PerformEditCommand (theCommand: longint);
```

```
void PerformEditCommand (long theCommand);
```

Perform the standard cut, copy, paste, and clear commands on the text. The task classes call this method to undo an edit command.

TypeChar

```
procedure TypeChar (theChar: char;
   theModifiers: integer);
```

```
void TypeChar (char theChar, short theModifiers);
```

Process a keystroke. This method does not need to set up for an **Undo** command and should handle the key directly. The task classes call this method to undo key strokes.

Display methods

Draw

```
procedure Draw (var area: Rect);
```

```
void Draw (Rect *area);
```

Draw the text pane.

Scroll

```
void Scroll (long hDelta, long vDelta,
   Boolean redraw);
procedure Scroll (hDelta, vDelta: longint;
   redraw: Boolean);
```

Scroll the text within the pane by `hDelta` characters and `vDelta` lines.

Activate

```
procedure Activate;
void Activate (void);
```

Activate the text pane. This method enables the editing commands and either highlights the selection or shows the text insertion caret.

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

Deactivate the text pane. This method disables the editing commands and either unhighlights the selection or hides the text insertion caret.

SetSelection

```
procedure SetSelection (selStart, selEnd: long;
   fRedraw: Boolean);
void SetSelection (long selStart, long selEnd,
   Boolean fRedraw);
```

Set the selection to the range corresponding to character positions `selStart` through `selEnd`.

GetSelection

```
procedure GetSelection (var selStart, selEnd:
   longint);
void GetSelection (long *selStart, long *selEnd);
```

Return the start and end of the current selection.

Text specification methods

SetTextPtr

```
procedure SetTextPtr (textPtr: Ptr;
   numChars: longint);
void SetTextPtr (Ptr textPtr, long numChars);
```

Use the first `numChars` characters that `textPtr` points to as the text for this abstract text object. This method makes a copy of the text.

GetTextHandle

```
function GetTextHandle: CharsHandle;
CharsHandle GetTextHandle (void);
```

Return a handle to the text of the text. This method returns a handle to the actual text, not to a copy of the text.

CopyTextRange

```
function CopyTextRange (start, end: long): Handle;
Handle CopyTextRange (long start, long end);
```

Return a copy of the range of text specified by `start` and `end`.

InsertTextPtr

```
procedure InsertTextPtr (text: Ptr; length: longint;
   fRedraw: Boolean);
void InsertTextPtr (Ptr text, long length,
   Boolean fRedraw);
```

Insert a copy of the given text and length at the start of the current selection. If `fRedraw` is TRUE, redraw the pane at the next update event.

CheckInsertion

```
procedure CheckInsertion (insertLen: longint;
   useSelection: Boolean);
void CheckInsertion (long insertLen,
   Boolean useSelection);
```

Check whether an insertion of `insertLen` characters would exceed TextEdit's capacity. If `useSelection` is TRUE, this method deducts the size of the selection from the total lenght. If the insertion would fail, this method calls `Failure`.

Text characteristics methods

SetFontNumber

```
procedure SetFontNumber (aFontNumber: integer);
void SetFontNumber (short aFontNumber);
```

Set the font for this text pane by font number.

SetFontStyle

```
procedure SetFontStyle (aStyle: Style);
void SetFontStyle (short aStyle);
```

Toggle the font style for this text pane. `AStyle` may be one of: `bold`, `italic`, `underline`, `outline`, `shadow`, `condense`, or `extend`

SetFontSize

```
procedure SetFontSize (aSize: integer);
void SetFontSize (short aSize);
```

Set the font size for this text pane to the specified size.

SetTextMode

```
procedure SetTextMode (aMode: integer);
void SetTextMode (short aMode);
```

Set the text mode for this text pane to the specified mode. `AMode` can be one of `srcOr`, `srcXor`, or `srcBic`.

SetAlignCmd

```
procedure SetAlignCmd (anAlignCmd: long);
void SetAlignCmd (long anAlignCmd);
```

Set the text alignment for this text pane. This method uses the THINK Class Library's names for the alignment choices: `cmdAlignLeft`, `cmdAlignRight`, or `cmdAlignCenter`.

GetAlignCmd

```
function GetAlignCmd: longint;
long GetAlignCmd (void);
```

Return the current alignment for this text pane. It can be one of `cmdAlignLeft`, `cmdAlignRight`, `cmdAlignCenter`, or `cmdNull`.

SetAlignment

```
procedure SetAlignment (anAlignment: long);
void SetAlignment (long anAlignment);
```

Set the text alignment for this text pane. This method uses TextEdit's names for the alignment choices: `teFlushDefault`, `teFlushLeft`, `teCenter`, or `teFlushRight`.

SetSpacingCmd

```
procedure SetSpacingCmd (aSpacingCmd: longint);
void SetSpacingCmd (long aSpacingCmd);
```

Set the space between lines of text. `ASpacingCmd` can be one of `cmdSingleSpace`, `cmd1HalfSpace`, or `cmdDoubleSpace`.

GetSpacingCmd

```
function GetSpacingCmd: longint;
long GetSpacingCmd (void);
```

Return the space between lines of text. It can be one of `cmdSingleSpace`, `cmd1HalfSpace`, or `cmdDoubleSpace`.

GetTEFontInfo

```
procedure GetTEFontInfo (var macFontInfo: FontInfo);
void GetTEFontInfo (FontInfo *macFontInfo);
```

Return a QuickDraw `FontInfo` record for the font that this text pane uses.

GetHeight

```
function GetHeight (startLine endLine: longint):
   longint;
long GetHeight (long startLine, long endLine);
```

Return the total height in pixels of the indicated lines of text.

GetCharOffset

```
function GetCharOffset (aPt: LongPt): longint;
long GetCharOffset (LongPt *aPt);
```

Return the character position nearest the coordinate `aPt`. `APt` must be in frame coordinates.

GetCharPoint

```
procedure GetCharPoint (offset: long;
   var aPt: LongPt);
void GetCharPoint( long offset, LongPt *aPt);
```

Return the coordinate of the character position `offset`. `APt` is in frame coordinates.

GetCharStyle

```
procedure GetCharStyle (charOffset: long;
   var theStyle: TextStyle);
void GetCharStyle (long charOffset,
   TextStyle *theStyle);
```

Return style information for the character at position `charOffset`.

GetTextStyle

```
procedure GetTextStyle (var whichAttributes: integer;
   var aStyle: TextStyle);
void GetTextStyle (short *whichAttributes,
   TextStyle *aStyle);
```

Return current style information for this text pane. `WhichAttributes` is a flag that indicates which text attributes to report on. The attributes flags and `TextStyle` record are described in *Inside Macintosh VI*, Chapter 15, "TextEdit"

Calibrating methods

ResizeFrame

```
procedure ResizeFrame (delta: Rect);
void ResizeFrame (Rect *delta);
```

Resize the static text's frame when the size of its pane changes. The `delta` rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left.

CalcTERects

```
procedure CalcTERects;
void CalcTERects (void);
```

Determine the destination and view rectangles for this text pane's TextEdit record.

AdjustBounds

```
procedure AdjustBounds;
void AdjustBounds (void);
```

Adjust the bounds of a this text pane to match its TextEdit record. When you do something that could change the line width or the number of lines, send this message.

FindLine

```
function FindLine (charPos: longint): longint;
short FindLine (long charPos);
```

Return the line number containing the specified character position. Both line and character numbering start at zero. If the character position is before the start of the text (a negative number), this method returns zero. If the character position is beyond the end of the text, this method return the number of the last line.

GetLength

```
function GetLength: longint;
long GetLength (void);
```

Return the length in bytes of this text pane's text buffer.

GetNumLines

```
function GetNumLines: longint
long GetNumLines (void);
```

Return the total number of lines in this text pane's text buffer.

Printing methods

AboutToPrint

```
procedure AboutToPrint (var firstPage, lastPage:
   integer);
void AboutToPrint (short *firstPage, short *lastPage);
```

The specified range of pages is about to be printed. This method deactivates the text pane to unhighlight the current selection.

PrintPage

```
procedure PrintPage (pageNum: integer;
   pageWidth, pageHeight: integer;
   aPrinter: CPrinter);
void PrintPage (short pageNum, short pageWidth,
   short pageHeight, CPrinter *aPrinter);
```

Print the specified page.

DonePrinting

```
procedure DonePrinting;
void DonePrinting (void);
```

Printing is over. This method re-highlights the current selection if the edit pane is active.

Cursor methods

Dawdle

```
procedure Dawdle (var maxSleep: longint);
void Dawdle (long *maxSleep);
```

This method flashes the insertion point when the edit text pane is active. This method sets `maxSleep` to the value of `GetCaretTime`, which is the

rate at which the insertion point blinks. Setting this value ensures that `WaitNextEvent` will generate a null event at least that often.

Note

To learn more about "sleep time," see the description of the `Dawdle` message in CBureaucrat page 216.

CEnvironment 33

Introduction

CEnvironment maintains a drawing environment for any pane.

Heritage

Superclass	CObject
Subclasses	CTextEnvironment

Using CEnvironment

Every pane has an `itsEnvironment` instance variable. If this variable points to an object of this class, the `Prepare` method sends it a `Restore` message to set up the QuickDraw drawing environment for the pane.

You can use this class to make sure that the drawing environment is set up correctly. Or you might want to change the drawing environment according to some saved settings.

Variables

This class has no instance variables.

Methods

Restore

```
procedure Restore;
void Restore (void);
```

Restore the drawing environment in the current port. The default method just calls the QuickDraw routine PenNormal.

CError 34

Introduction

CError is an error-handling class. You can use the global error handler to report errors.

Note

This class is included for compatibility with earlier versions of the THINK Class Library. New programs that you write with the THINK Class Library should use the exception handling mechanism described on page 127.

Heritage

Superclass	CObject
Subclasses	None

Using CError

The application initialization method `IApplication` creates an instance of CError and stores it in the global variable `gError`. Several objects in the THINK Class Library use the global error handler to post error messages. The default error handler reports messages and gives you a chance to quit or to proceed with the application. Create a subclass of this method to implement more sophisticated error recovery.

Variables

Global variable

Variable	Type	Description
`gError`	`CError`	Global error handler.

Instance variables

This class has no instance variables.

Methods

Error reporting methods

SevereMacError

```
procedure SevereMacError (macErr: OSErr);
void SevereMacError (OSErr macErr);
```

Report an operating system error in a Stop Alert that gives you a chance to quit the application or jump to the beginning of the event loop. This method looks for an `Estr` resource with the same ID as an operating system error. If it doesn't find one, it uses the default string (`STR 200`). If you press the Quit button, this method sends a `Quit` message to the application. If you press the Proceed button, this method sends the application a `JumpToEventLoop` message.

CheckOSError

```
function CheckOSError (macErr: OSErr): Boolean;
Boolean CheckOSError (OSErr macErr);
```

Check for a Macintosh operating system error. Return `TRUE` if everything is OK, `FALSE` otherwise. If `macErr` is an error, this method displays a Stop Alert. This method looks for an `Estr` resource with the same ID as the operating system error. If it can't find one, it uses a default string.

PostAlert

```
procedure PostAlert (STRid: integer; index: integer);
void PostAlert (short STRid, short index);
```

Post a general alert and return. `STRid` is the resource ID of a `STR#` resource. `Index` is the index into the `STR#` resource. The string is displayed in a generic alert box.

MissingResources

```
procedure MissingResources;
void MissingResources (void);
```

Post an alert announcing that the application can't find its resources, and exit the application. The alert suggests that your project doesn't have an associated resource file. The alert can't be in a resource, since this method is called only if the resource file is unavailable.

Functions

Note that these are functions and procedures, not methods.

GrowZoneFunc

```
function GrowZoneFunc (bytesNeeded: Size): longint;
pascal long GrowZoneFunc (Size bytesNeeded);
```

The application method `InitMemory` installs this function as the application's `GrowZone` function which is called in low memory conditions. This function sends a `GrowMemory` message to the application. See CApplication on page 163.

CheckResource

```
procedure CheckResource (r: Handle);
void CheckResource (Handle r);
```

If the handle `h` is `NIL`, this procedure sends the message `SevereMacError(resNotFound)` to the global `gError`. Call this function after trying to retrieve a resource.

CheckAllocation

```
procedure CheckAllocation (p: Ptr);
void CheckAllocation (void *p);
```

If `p` is `NIL`, the procedure calls `SevereMacError(MemError)` to report a severe memory error. Call this function after trying to allocate memory.

Class resources

Resource	Description
`ALRT/DITL 128`	Generic alert box. Contains `^0` for use with `ParamText`.
`ALRT/DITL 200`	Alert box for a `SevereError`
`ALRT/DITL 300`	Alert box for a Macintosh operating system error.
`STR 300`	String that reports a severe Macintosh system error
`Estr` resources	An `Estr` resource has the same format as a `STR` resource. The `SevereMacError` method looks for an `Estr` resource with the same ID as an operating system error. For instance, `Estr -42` might read "Too many files open."

CFile 35

Introduction

CFile is an abstract class for implementing classes that deal with disk files.

Heritage

Superclass	CObject
Subclasses	CDataFile
	CResFile

Using CFile

CFile is an abstract class for dealing with Macintosh disk files. Most of the time you'll use the CDataFile subclass to work with regular data files.

Before you open a file, you must specify it. Specifying means identifying it to the Macintosh file manager. This class gives you three ways to specify a file depending on the type of information you can provide. The most common specification method, `SFSpecify`, lets you use an `SFReply` record from the standard file dialogs to identify a file.

Some CFile methods use the THINK Class Library's exception handling library if something goes wrong in a file operation. For more information on exception handling, see Chapter 8, "Exception Handling.

Note

This version of CFile handles errors differently than the version included with THINK C 4.0. The old version is now named OFile and is in the `Compatibility Classes` folder.

Variables

Variable	Type	Description
name	Str63	File name
volNum	integer	Volume containing the file
dirID	longint	Directory within the volume

Methods

Construction and destruction methods

IFile

```
procedure IFile;
void IFile (void);
```

Initialize the file object.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Close and dispose of the file object.

Specifying methods

Specify

```
procedure Specify (aName: Str63; aVolNum: integer);
void Specify (Str63 aName, short aVolNum);
```

Specify a file by its name and volume reference number. Use this method to specify files on MFS volumes.

SpecifyHFS

```
procedure SpecifyHFS (aName: Str63; aVolNum: integer;
   aDirID: longint);
void SpecifyHFS (Str63 aName, short aVolNum,
   long aDirID);
```

Specify a file by its name, volume number, and directory ID. If the specification is for an alias, this method resolves it.

SpecifyFSSpec

```
procedure SpecifyFSSpec (aFileSpec: FSSpec);
void SpecifyFSSpec (const FSSpec *aFileSpec)
```

Specify the File using a File Manager FSSpec record. If the specification is for an alias, this method resolves it.

SFSpecify

```
procedure SFSpecify (macSFReply: SFReplyPtr);
void SFSpecify (SFReply *macSFReply);
```

Specify a file from the information in a macSFReply record. If the specification is for an alias, this method resolves it. Use this method to specify a file that the user chose through a standard file dialog.

ResolveFileAlias

```
procedure ResolveFileAlias;
void ResolveFileAlias (void)
```

If the file specification is an alias, resolve it.

Open and close methods

Open

```
procedure Open (permission: SignedByte);
void Open (SignedByte permission);
```

Open the file with the specified permission. This method does nothing. Subclasses must override this method. For an example of how to write an Open method, see CDataFile and other descendants of this class.

Close

```
procedure Close;
void Close (void);
```

Close this file. This method does nothing. Subclasses must override this method.

Accessing methods

GetName

```
procedure GetName (var theName: Str63);
void GetName (Str63 theName);
```

Get the name of the file.

GetFSSpec

```
procedure GetFSSpec (var aFileSpec: FSSpec);
void GetFSSpec (FSSpec *aFileSpec)
```

Get an FSSpec record for this file.

ExistsOnDisk

```
function ExistsOnDisk: Boolean;
Boolean ExistsOnDisk (void);
```

Returns TRUE if there's an existing file that matches the current specification.

GetMacFileInfo

```
procedure GetMacFileInfo (var fileInfo: FInfo)
void GetMacFileInfo (FInfo *fileInfo);
```

Return the Finder information for this file. The information includes the file's type, creator, and icon position.

Filing methods

CreateNew

```
procedure CreateNew (creator, fType: OSType);
void CreateNew (OSType creator, OSType fType);
```

Create new file with the specified creator and file type. This method uses the name and volume information you set up with one of the specification

methods. You can use the application signature in `gSignature` for the creator.

ThrowOut

```
procedure ThrowOut;
void ThrowOut (void);
```

Close the file and delete it.

ChangeName

```
procedure ChangeName (newName: Str255);
void ChangeName (Str63 newName);
```

Give this file a new name.

CFWDesktop 36

Introduction

CFWDesktop is a subclass of CDesktop that supports floating windows.

Heritage

Superclass	CDesktop
Subclasses	None

Using CFWDesktop

CFWDesktop is a subclass of CDesktop that supports floating windows. In general, your application shouldn't need to send messages to the desktop.

If you use CFWDesktop, be sure to override `MakeDesktop` to create a floating window desktop and store the object in `gDesktop`.

This is how you do it in Pascal:

```
procedure CMyApp.MakeDesktop;
begin
   new(CFWDesktop(gDesktop));
   CFWDesktop(gDesktop).IFWDesktop(SELF);
end;
```

And this is how you do it in C:

```
void CMyApp::MakeDesktop(void)
{
   gDesktop = new(CFWDesktop);
   ((CFWDesktop*)gDesktop)->IFWDesktop(this);
}
```

To make a floating window, just pass `TRUE` as the `aFloating` parameter to `IWindow`, CWindow's intialization method.

Variables

Variable	Type	Desktop
itsFloats	CList	List of floating windows
topFloat	CWindow	Topmost floating window

Methods

Construction and destruction methods

IFWDesktop

```
procedure IFWDesktop (aSupervisor: CBureaucrat);
void IFWDesktop (CBureaucrat *aSupervisor);
```

Initialize the desktop. This method calls CDesktop's initialization method and creates the list of floating window..

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the desktop and send this message to all of the desktop's windows.

Appearance methods

Show

```
procedure Show;
void Show (void);
```

Makes a desktop visible by showing all of its windows. This method sends a Show message to all of the desktop's windows and floating windows.

Hide

```
procedure Hide;
void Hide (void);
```

Hide a desktop by hiding all of its windows. This method sends a Hide message to all of the desktop's windows and floating windows.

Activate

```
procedure Activate;
void Activate (void);
```

Activate the desktop by activating all the windows. This method sends an Activate message to the desktop's top window and to all of the floating windows.

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

Deactivate a desktop by deactivating all the windows. This method sends a Deactivate message to the desktop's top window and to all of the floating windows.

Mouse methods

AdjustCursor

```
procedure AdjustCursor (where: Point;
   mouseRgn: RgnHandle);
void AdjustCursor (Point where, RgnHandle mouseRgn);
```

Adjust the cursor shape when the mouse is in an insignificant part of the desktop. The default method sets the cursor to an arrow.

Window methods

You do not have to use any of these methods yourself. Directors, objects that manage the interaction between windows and the desktop, take care of sending most of these messages.

AddWind

```
procedure AddWind (theWindow: CWindow);
void AddWind (CWindow *theWindow);
```

Add theWindow to the desktop's window list. If the window is a floating window, add it to the beginning of the itsFloats list.

RemoveWind

```
procedure RemoveWind (theWindow: CWindow);
void RemoveWind (CWindow *theWindow);
```

Remove theWindow from the desktop's window list. If the window is a floating window, remove it from the itsFloats list.

SelectWind

```
procedure SelectWind (theWindow: CWindow);
void SelectWind (CWindow *theWindow);
```

Select theWindow and bring theWindow to the front. It the window is a floating window, bring it to the front of the itsFloats list and mark it as the topFloat window.

ShowWind

```
procedure ShowWind (theWindow: CWindow);
void ShowWind (CWindow *theWindow);
```

Make theWindow visible on the desktop.

HideWind

```
procedure HideWind (theWindow: CWindow);
void HideWind (CWindow *theWindow);
```

Hide theWindow.

DragWind

```
procedure DragWind (theWindow: CWindow;
   macEvent: EventRecord);
void DragWind (CWindow *theWindow,
   EventRecord *macEvent);
```

Drag `theWindow` on the desktop. This method does not use `DragWindow` Toolbox routine.

CalTopFloat

```
procedure CalcTopFloat;
void CalcTopFloat (void);
```

Set `topFloat` to the first visible floating window. Send this message to the desktop if you hide or remove a floating window.

Cleanup

Cleanup method

```
procedure Cleanup;
```

This method is for THINK Pascal. When you're running in the THINK Pascal environment, all the windows share the same space. As you debug your program and open source windows and debugging windows, the window order changes. When you resume your program, THINK Pascal will move your application's active window to the front, but it doesn't know about floating windows. This method moves all of THINK Pascal's windows to the back, and cleans up your application windows so the floating windows float and the application windows are in the right order.

CGridSelector 37

Introduction

CGridSelector is an abstract class for drawing a pane with several items arranged in a table.

Heritage

Superclass	CSelector
Subclasses	CCharGrid
	CPatternGrid

Using CGridSelector

CGridSelector is a descendant of CSelector that lets you display items in a table. To use CGridSelector, all you have to do in your subclass is override the `DrawItem` method. CGridSelector takes care of everything else.

The THINK Class Library includes two subclasses of CGridSelector. CPatternGrid displays a pattern palette. CCharGrid displays a set of characters. You can use CCharGrid with a special font to display tool palettes.

Items in a grid are numbered from 1, row first.

Variables

Variable	Type	Description
`rows`	`integer`	The number of rows in the grid.
`columns`	`integer`	The number of columns in the grid.
`boxWidth`	`integer`	The width in pixels of each box.
`boxHeight`	`integer`	The height in pixels of each box.
`gridOn`	`Boolean`	True if the grid lines should be drawn.

Methods

Construction methods

IGridSelector

```
procedure IGridSelector (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption;
   aSelection, aCommandBase: integer
   aRows, aColumns, aBoxWidth, aBoxHeight: integer);
```

```
void IGridSelector (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aSelection, short aCommandBase
   short aRows, short aColumns,
   short aBoxWidth, short aBoxHeight);
```

Initialize a selector. The first eight arguments to this routine are identical to the ones for `IPane`.

See the description of CSelector's DoClick method on page 421 to learn how selectors generate command numbers.

`ASelection` is the initial item to select. `ACommandBase` is the base value for converting the selected item into a command number. `ARows` and `aColumns` determine how many boxes are in the gird. `ABoxWidth` and `aBoxHeight` determine the size of each box in pixels. Finally, `IGridSelector` sets the vertical and horizontal scales to the size of the boxes.

Note

The descriptions of the other arguments are in CPane on page 347.

Drawing methods

Draw

```
procedure Draw (var area: Rect);
```

```
void Draw (Rect *area);
```

Draws the grid selector. If `gridOn` is true, this method first draws the grid. Then it uses `DrawItem` to draw each box. Finally, it highlights the current selection.

Since most of the work is done in the `DrawItem` method, you probably won't need to override this method.

DrawGrid

```
procedure DrawGrid;
```

```
void DrawGrid (void);
```

Draw the grid lines between items. The `Draw` method calls `DrawGrid` if `gridOn` is TRUE.

DrawItem

```
procedure DrawItem (theItem: integer;
   theBox: Rect);
```

```
void DrawItem (short theItem, Rect *theBox);
```

Draw the item. `TheItem` is the number of the item to draw, and `theBox` is the rectangle to draw it in. Your subclass must override this method.

Accessing methods

HiliteItem

```
procedure HiliteItem (theItem: integer;
   state: HiliteState);
```

```
void HiliteItem (short theItem, HiliteState state);
```

Hilight the specified item. `HiliteItem` can take on of three values for the `state` parameter:

Hilite state value	Behavior
`hiliteOFF`	Invert the item's box.
`hiliteON`	Invert the item's box.
`hiliteDYNAMIC`	Flash the edges of the item's box.

If you want to higlight an item differently, you can override this method.

FindItem

```
function FindItem (hitPt: Point): integer;
```

```
short FindItem (Point hitPt);
```

Determine which item corresponds to a mouse down at a specified point. If the `hitPt` isn't in the grid, `FindItem` returns zero. Grid selector items are numbered from 1, row first like this:

1	2	3
4	5	6
7	8	9

Figure 37-1 Item numbering in grid selectors

FindItemBox

```
procedure FindItemBox (theItem: integer;
   var theBox: Rect);
short FindItemBox (short theItem, Rect *theBox);
```

Return the box that corresponds to `theItem`. `HiliteItem` uses this method to figure our what box to highlight.

SetGridOn

```
procedure SetGridOn (aGridOn: Boolean);
void SetGridOn (Boolean aGridOn);
```

Sets the instance variable `gridOn` to `aGridOn`. If `gridOn` is true, the `Draw` method calls `DrawGrid` to draw grid lines between items.

CList 38

Introduction

CList implements an ordered list of objects.

Heritage

Superclass	CCluster
Subclasses	None

Using CList

Use an object of class CList when you need to maintain an ordered list of objects. Several of the objects in the THINK Class Library use CList to maintain lists. You can use the iteration methods inherited from CCluster to apply functions to each item in a list. Every object in a list has an index. Index values begin at 1.

Variables

This class has no instance variables.

Methods

For methods that insert new objects before or after existing objects, be sure that the existing object is actually in the list. If you don't, strange things may happen.

Construction methods

CList inherits the `Free` (in THINK Pascal) or `Dispose` (in THINK C) method from CCluster.

IList

```
procedure IList;
void IList (void);
```

Initialize the list.

Insertion and deletion methods

Append

```
procedure Append (theObject: CObject);
void Append (CObject *theObject);
```

Add `theObject` to the end of the list.

Prepend

```
procedure Prepend (theObject: CObject);
void Prepend (CObject *theObject);
```

Add `theObject` to the beginning of the list.

InsertAfter

```
procedure InsertAfter (theObject: CObject;
   afterObject: CObject);
void InsertAfter (CObject *theObject,
   CObject *afterObject);
```

Insert the `theObject` after the object specified by `afterObject`.

InsertAt

```
procedure InsertAt (theObject: CObject;
   index: longint);
void InsertAt (CObject *theObject, long index);
```

Make `theObject` be the `index`th object in the list. All subsequent objects move down one position

Ordering methods

BringFront

```
procedure BringFront (theObject: CObject);
void BringFront (CObject *theObject);
```

Make `theObject` be the first object in the list.

SendBack

```
procedure SendBack (theObject: CObject);
void SendBack (CObject *theObject);
```

Make `theObject` be the last object in the list.

MoveUp

```
procedure MoveUp (theObject: CObject);
void MoveUp (CObject *theObject);
```

Move `theObject` up one slot toward the front of the list. The object that was before it in the list moves down one slot.

MoveDown

```
procedure MoveDown (theObject: CObject);
void MoveDown (CObject *theObject);
```

Move `theObject` down one slot toward the end of the list. The object that was after it in the list moves up one slot.

MoveToIndex

```
procedure MoveToIndex (theObject: CObject;
   index: longint);
void MoveToIndex (CObject *theObject, long index);
```

Move `theObject` to the `index`th position in the list. If `theObject` moves back in the list, the items between its original position and its new position move up one slot. If `theObject` moves forward in the list, the items between its new position and its original position move down one slot.

Membership methods

FirstItem

```
function FirstItem: CObject;
CObject* FirstItem (void);
```

Return the first item in the list.

LastItem

```
function LastItem: CObject;
CObject* LastItem (void);
```

Return the last item in the list.

NthItem

```
function NthItem (n: longint): CObject;
CObject* NthItem (long n);
```

Return the nth item in the list.

FindIndex

```
function FindIndex (theObject: CObject): longint;
long FindIndex (CObject *theObject);
```

Return the index of the object. Indexes begin at 1. If the item is not in the list, this method returns 0.

FirstSuccess

```
function FirstSuccess (function testFunc(theObject:
   CObject): Boolean): CObject;
CObject* FirstSuccess (TestFunc testFunc);
```

Return the first item that causes the `testFunc` function to return `TRUE`.

In THINK C, `testFunc` must be declared like this:

```
Boolean MyTestFunc (CObject *theObject);
```

FirstSuccess1

```
function FirstSuccess1 (function testFunc (theObject:
   CObject; theParam: Ptr): Boolean; param: Ptr):
   CObject;
CObject* FirstSuccess1 (TestFunc1 testFunc,
   long param);
```

Return the first item that causes the `testFunc` function to return `TRUE`.

In THINK C, `testFunc` must be declared like this:

```
Boolean MyTestFunc (CObject *theObject,
   long param);
```

LastSuccess

```
function LastSuccess (function testFunc(theObject:
   CObject): Boolean): CObject;
CObject* LastSuccess (TestFunc testFunc);
```

Return the last item in the list that causes the `testFunc` function to return `TRUE`.

In THINK C, `testFunc` must be declared like this:

```
Boolean MyTestFunc (CObject *theObject);
```

LastSuccess1

```
function LastSuccess1 (function testFunc(theObject:
   CObject; theParam: Ptr): Boolean; param: Ptr):
   CObject;
CObject* LastSuccess1 (TestFunc1 testFunc,
   long param);
```

Return the last item in the list that causes the `testFunc` function to return `TRUE`.

In THINK C, `testFunc` must be declared like this:

```
Boolean MyTestFunc (CObject *theObject,
   long param);
```


CMBarChore 39

Introduction

CMBarChore is a chore that works with CBartender to redraw the menu bar.

Heritage

Superclass	CChore
Subclasses	None

Using CMBarChore

The CBartender class uses a CMBarChore to redraw the menu bar after deleting menus from the menu bar. Your application should not need to use this class.

The CBartender's `DeleteFromBar` method creates a CMBarChore and assigns it as an urgent chore, so the menu bar gets redrawn the next time through the event loop.

Variables

This class has no instance variables.

Methods

Perform

```
procedure Perform (var maxSleep: Longint);
void Perform (long *maxSleep);
```

This method calls the Toolbox routine `DrawMenuBar` to redraw the menu bar.

CMenuDefProc 40

Introduction

CMenuDefProc is an abstract class that gives you an object-oriented interface for Macintosh menu definition procedures (MDEFs). The description of this class assumes that you're familiar with the Macintosh Menu Manager in general and with MDEFs in particular. If you need to learn more, see *Inside Macintosh I*, Chapter 11, "The Menu Manager" and *Inside Macintosh V*, Chapter 13, "The Menu Manager."

Heritage

Superclass	CObject
Subclasses	CPaneMDEF

Using CMenuDefProc

CMenuDefProc lets you write object-oriented menu definition functions. To write an MDEF object, you need to create a subclass of CMenuDefProc or use one of the ones provided in the THINK Class Library.

CMenuDefProc uses a stub MDEF that contains a jump to a generic menu definition procedure and a reference to an MDEF object. The Menu Manager calls the generic menu definition procedure which dispatches messages to your MDEF object.

Creating the stub MDEF

The stub MDEF is a 10-byte data structure that looks like this in Pascal:

```
GenericMDEFRec = record
   JMPinstruction: integer;
   defProc: ProcPtr;
   itsMenuDefProc: CMenuDefProc;
end;
```

And in C it looks like this:

```
typedef struct GenericMDEFRec {
   short          JMPinstruction;
   VoidFunc       defProc;
   CMenuDefProc*  itsMenuDefProc;
};
```

To create the MDEF, use your favorite resource editor or resource compiler to create a 10-byte resource that contains these hex values:

```
4EF9 0000 0000 0000 0000
```

When you create the menu that uses your MDEF, be sure to specify the MDEF ID. Otherwise, the Menu Manager uses the default MDEF.

Apple reserves IDs 0 to 127 for definition procedures and recommends that you number your MDEF in the range 128 to 4095. In the THINK Class Library, the convention is to give your MDEF the same ID as the menu that you'll be using it with.

Writing the CMenuDefProc subclass

Your object-oriented MDEF must handles all the messages that a regular MDEF handles. That means that your subclass needs to override all the methods of CMenuDefProc:

Initialization	DrawMenu	ChooseItem
SizeMenu	PlacePopup	

Your subclass can define instance variables to hold whatever information your menu needs to do its work. Your initialization method should allocate memory for the instance variables if necessary. Your subclass must call IMenuDefProc to set up the stub MDEF resource.

If you want to use your MDEF for more than one menu, be careful how you use instance variables. If the instance variables of the MDEF control the state of the menu being displayed, you'll only be able to use it for one menu.

For example, the CPaneMDEF subclass of this class keeps two instance variables to refer to the pane being displayed as a menu and to the tear-off menu. If you were to use the same MDEF based on CPaneMDEF for two menus, both menus would display the same pane.

Using your CMenuDefProc subclass

The most important thing to keep in mind about using a descendant of CMenuDefProc is that you need to initialize it before the Menu Manager loads the menu associated with it. If you don't, your application will crash.

The reason that it's important to initialize the custom MDEF object before CApplication's SetUpMenus or CBartender's AddMenu has to do with the way the Menu Manager works and the way IMenuDefProc does its job.

The description for GenericMDEF is near the end of this chapter on page 334

IMenuDefProc loads the MDEF and sets up the fields so defProc points to a function called GenericMDEF, and the itsMenuDefProc points to the current object. GenericMDEF is the "real" menu definition procedure that dispatches Menu Manager messages to your MDEF object.

As it's loading the menu, the Toolbox routine GetMenu loads the MDEF and sends it an mSizeMsg to find out how big the menu is. If you haven't initialized the MDEF, this call to menu definition procedure will jump to random memory, and your program will crash.

Examples of MDEF objects

If you're including the menu in your application's MBAR resource, override CApplication's SetUpMenus method to create the MDEF object and initialize it. Then call CApplication's SetUpMenus method. Chances are that you're overriding this method anyway to load resource-based menus like the Font menu and to set up the dimming and checking options for your menus.

This is what your SetUpMenus method should look like in Pascal:

```
procedure CMyApp.SetUpMenus;
var
   theMDEF: CMyCustomMDEF;
begin
   new(theMDEF);
   theMDEF.IMyCustomMDEF(CustomID);
   inherited SetUpMenus;
   { load font menu, set check & dim options }
end;
```

In C, it should look like this:

```
void CMyApp::SetUpMenus (void)
{
   CMyCustomMDEF theMDEF;

   theMDEF = new(CMyCustomMDEF);
   theMDEF->IMyCustomMDEF(CustomID);
   inherited::SetUpMenus();
   /* load font menu, set check & dim options */
}
```

Your initialization method (IMyCustomMDEF in this example) must call CMenuDefProc's IMenuDefProc.

If you're using CBartender's AddMenu method to add the menu to the menu bar, you need to create and initialize the MDEF object before you call AddMenu. The next example shows a method that creates a tool menu that uses a custom MDEF and adds it to the end of the menu bar. Note that the example uses the TCL convention of giving the menu and MDEF the same ID.

This is what the method looks like in Pascal:

```
procedure CPaintPane.AddToolMenu;
var
   theToolMDEF: CToolMDEF;
begin
   new(theToolMDEF);
   theToolMDEF.IToolMDEF(ToolMENUID);
   gBartender.AddMenu(ToolMENUID, TRUE, 0);
end;
```

In it looks like this:

```
void CPaintPane::AddToolMenu(void)
{
   CToolMDEF theToolMDEF;

   theToolMDEF = new(CToolMDEF);
   theToolMDEF->IToolMDEF(ToolMENUID);
   gBartender->AddMenu(ToolMENUID, TRUE, 0);
}
```

Variables

This class has no instance variables.

Methods

Except for the initialization method, all of CMenuDefProc's methods handle a Menu Manager message to a menu definition procedure. The descriptions given here of what your subclass should do are fairly complete, but for details, be sure to consult *Inside Macintosh I*, Chapter 11, "The Menu Manager" and *Inside Macintosh V*, Chapter 13, "The Menu Manager."

IMenuDefProc

```
procedure IMenuDefProc (MDEFid: Integer);
void IMenuDefProc (short MDEFid);
```

Initialize the object. This method loads the stub MDEF and stores the address of the generic MDEF function GenericMDEF in the defProc field. It stores the reference to this object in the itsMenuDefProc field.

DrawMenu

```
procedure DrawMenu (macMenu: MenuHandle;
   menuRect: Rect);
void DrawMenu (MenuHandle macMenu, Rect *menuRect);
```

Draw the menu in response to a Menu Manager mDrawMsg. MenuRect is given in global coordinates, the current port is the Window Manager port, and the clipping region is set to menuRect. Your DrawMenu method should check whether the menu is enabled and draw it in gray if it's not.

ChooseItem

```
procedure ChooseItem (macMenu: MenuHandle;
   menuRect: Rect; hitPt: Point;
   var whichItem: Integer);
void ChooseItem(MenuHandle macMenu, Rect *menuRect,
   Point hitPt, short *whichItem);
```

Choose an item in response to a Menu Manager mChooseMsg. Both hitPt and menuRect are in global coordinates, and whichItem is the last item chosen. WhichItem is initially 0. Your ChooseItem method should check that hitPt is within menuRect, and if it is, it should unhighlight whichItem and highlight the new item (if it's not the same as whichItem), then set whichItem to the new item. If hitPt isn't in menuRect, you should unhighlight whichItem and set whichItem to 0.

SizeMenu

```
procedure SizeMenu (macMenu: MenuHandle);
void SizeMenu (MenuHandle macMenu);
```

Set the size of the menu in response to a Menu Manager mSizeMsg. Your method should store the height of the menu in macMenu's menuWidth field and the height in the menuHeight field. Note that the menu manager sends this message to your menu soon after it has been loaded. See "Using your CMenuDefProc subclass" on page 330 for more details.

PlacePopUp

```
procedure PlacePopUp (macMenu: MenuHandle;
   menuRect: Rect; hitPt: Point;
   var whichItem: integer);
void PlacePopUp (MenuHandle macMenu, Rect *menuRect,
   Point hitPt, short *whichItem);
```

Determine the rectangle for a pop-up menu in response to a Menu Manager mPopUpMsg. WhichItem contains the previously selected item, and hitPt contains the top left corner of the pop-up menu (hitPt.h contains

the top and `hitPt.v` contains the left). Your method should set `menuRect` to the rectangle the menu is displayed in. If the menu would scroll, set `whichItem` to the actual top item. *Inside Macintosh V*, Chapter 13, "The Menu Manager" contains additional informati on you should be aware of.

Functions

CMenuDefProc uses an auxiliary function, `GenericMDEF`, to convert messages from the Menu Manager to object-oriented messages. In THINK C, this function is in `GenericMDEF.c` in the `FW/Tearoffs` folder, and in THINK Pascal it's in `GenericMDEF.p` in the `FW/Tearoffs` folder.

GenericMDEF

```
procedure GenericMDEF (theMesage: integer;
   macMenu: MenuHandle;
   menuRect: Rect; hitPt: Point;
   var whichItem: integer);

pascal void GenericMDEF (short theMessage,
   MenuHandle macMenu, Rect *menuRect,
   Point hitPt, short *whichItem);
```

As far as the Menu Manager is concerned, `GenericMDEF` is the "real" menu definition procedure. `GenericMDEF`. This routine dispatches messages to your object-oriented `MDEF`.

Be aware that the GenericMDEF only handles the four standard Menu Manager messages. If you write an `MDEF` that responds to user messages (as described in *Inside Macintosh V*, Chapter 13, "The Menu Manager"), you will need to write your own version of `GenericMDEF` and write your initialization method accordingly.

CMouseTask 41

Introduction

CMouseTask is an abstract class that implements mouse tracking.

Heritage

Superclass	CTask
Subclasses	You must define a subclass of this class.

Using CMouseTask

CMouseTask is an abstract class that lets you implement undoable mouse-related actions. For example, if you're writing a drawing application, you want to make sure that you can undo anything that you move or draw.

You don't have to use this class to implement mouse-related actions. You can always track the mouse yourself in your pane's `DoClick` method. If you do use CMouseTask to implement mouse actions, you don't have to make them undoable.

Defining a mouse task

To implement a mouse tracking task, define a subclass of CMouseTask and override the `KeepTracking` and `EndTracking` methods. The `KeepTracking` method does whatever you want to happen while the mouse is down. The `EndTracking` method does whatever you want to happen when the mouse is released.

For example, if you're moving a rectangle from one place in a pane to another, the `KeepTracking` method might draw a gray outline that moves as you move the mouse. The `EndTracking` method would erase the rectangle from its old location and redraw it in the new location.

If you want to make your mouse task undoable, you need to store enough information in the object to undo the effects of mouse tracking. You must

also override the Undo method (inherited from CTask) to use this information to undo the effects of the mouse task.

Using the moving rectangle example again, your EndTracking method might keep the old location of the rectangle in an instance variable. The Undo method would erase the rectangle from its current location and redraw it again in the old location.

Using the mouse task

You use mouse tasks in the DoClick method of a pane. Create an instance of your mouse task, initialize it, and pass it in a TrackMouse message to your pane. This is what part of your DoClick method might look like in THINK Pascal:

```
var
   myMover: CShapeMover;
begin
   new(myMover );
   myMover.IShapeMover(UNDOMoverStr);

   TrackMouse(myMover, hitPt, pinRect);
   itsSupervisor.Notify(myMover);
   ...
end
```

And this is what part of your DoClick method might look like in THINK C:

```
CShapeMover *myMover;

myMover = new(CShapeMover);
myMover->IShapeMover(UNDOMoverStr);

this->TrackMoouse(myMover, hitPt, &pinRect);
itsSupervisor->Notify(myMover);
```

The TrackMouse method sends your mouse task a BeginTracking message to give you an opportunity to adjust the starting point. As long as the mouse button is down, TrackMouse repeatedly sends KeepTracking messages to your mouse task. When the mouse button is released, TrackMouse sends your mouse task an EndTracking message.

The value you pass to your initialization method is the index of a string in the STR# 130 resource that describes your task. The document method UpdateUndo uses this string for the wording of the **Undo** command in the **Edit** menu. If your task is not undoable, you can use any value and ignore it.

After you've tracked the mouse, you can send the task in a Notify message to the document. The document will store the task in the document's lastTask instance variable. When you choose **Undo** from the **Edit** menu, the document sends an Undo message to the task to undo the effects of mouse tracking.

Variables

This class has no instance variables.

Methods

Construction and destruction methods

IMouseTask

```
procedure IMouseTask (aNameIndex: integer);
void IMouseTask (short aNameIndex);
```

Initialize a mouse tracking task. ANameIndex is the index for the undo/redo string in the STR# 130 resource.

Mouse tracking methods

BeginTracking

```
procedure BeginTracking (var startPt: LongPt);
void BeginTracking (LongPt *startPt);
```

Mouse tracking is starting. The document's TrackMouse method sends this message at the beginning of mouse tracking. StartPt is the starting point (in local coordinates). If your mouse task subclass doesn't alter the starting point, you don't need to override this method.

KeepTracking

```
procedure KeepTracking (var currPt, prevPt, startPt:
   LongPt);
void KeepTracking (LongPt *currPt, LongPt *prevPt,
   LongPt *startPt);
```

Mouse tracking is under way. The document's TrackMouse method sends this message repeatedly as long as the mouse button is down. CurrPt is the current mouse location. PrevPt is the previous mouse location. StartPt is a the original mouse location. Your mouse task subclass must override this method.

EndTracking

```
procedure EndTracking (var currPt, prevPt, startPt:
   LongPt);
void EndTracking (LongPt *currPt, LongPt *prevPt,
   LongPt *startPt);
```

Mouse tracking is over. The document's TrackMouse method sends this message when the mouse button is released. CurrPt is the current mouse location. PrevPt is the previous mouse location. StartPt is the original

mouse location. Your mouse task subclass must override this method. If your mouse task is undoable, you should store all the information you need to undo it in this method.

Note

If you're implementing an undoable mouse task, you'll need to override the Undo method as well.

Class resources

Resource	Description
STR# 130	List of strings that describe undoable tasks. For example, if you're implementing a mouse task that moves graphic images, the string for that task might be "move." The item in the **Edit** menu would then read "Undo move" or "Redo move."

CObject 42

Introduction

CObject is the abstract root level class. It is the ancestor of all the classes in the THINK Class Library.

Heritage

Superclass: None

Subclasses:

- CAppleEvent
- CBartender
- CBitMap
- CChore
- CCollaborator
- CDecorator
- CEnvironment
- CError
- CFile
- CMenuDefProc
- CPaneBorder
- CPrinter
- CSwitchboard
- CTask

Using CObject

Every class in the THINK Class Library is a descendant of this class. CObject is the only class with no superclass. If you need to create a new class that can't be a subclass of any other class, make it a subclass of CObject.

Variables

This class has no instance variables.

Methods

All classes inherit these methods:

Creation and destruction

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the memory an object occupies. If your subclass allocates memory in its initialization method, be sure you release it in this method.

Clone/Copy

```
function Clone: CObject;
CObject *Copy (void);
```

Make a copy of the object. Note that if a class has instance variables that point to a block of memory, only the reference is copied, not the contents of the block of memory.

Lock

```
function Lock (fLock: Boolean): Boolean;
Boolean Lock (Boolean fLock)
```

When `fLock` is TRUE, make this object non-relocatable. When `fLock` is FALSE, make it relocatable. This method returns TRUE if the object was locked.

SubclassResponsibility

```
procedure SubclassResponsibility;
void SubclassResponsibility (void);
```

In some cases, certain methods inherited from abstract classes must be overridden, or the subclass won't work correctly. Most of these methods call `SubclassResponsibility` rather than leaving the method empty. When the symbol `__TCL_DEBUG__` is defined, this method displays a debugging message.

GetClassName

```
procedure GetClassName (var className: Str255);
void GetClassName (Str255 className);
```

Place the name of this class in `className`.

CPane 43

Introduction

CPane is an abstract class that defines a drawing area within a window or within another pane. In the THINK Class Library, all drawing is done within a pane. Each pane has its own drawing environment and coordinate system.

Heritage

Superclass	CView
Subclasses	CControl
	CPanorama
	CRadioGroupPane
	CScrollPane
	CSizeBox

Using CPane

Use a pane when you need a non-scrolling area to draw in within a window. If you need a scrollable area, use CPanorama and CScrollPane. If you need a pane for displaying text, see CAbstractText or CEditText.

Your application must define a subclass of CPane (or one of its descendants) to draw in a window. In your subclass, you need to override or define these methods:

Initialization method	`Free`
`Draw`	`DoClick`

Your pane class should have an initialization method. If your subclass defines new instance variables, this is the method that sets them up. By convention, the name of your initialization method should be `I` *YourPane* where *YourPane* is the name of your pane class. Your initialization method should call `IPane`. The supervisor of a pane should be either the pane that encloses it or the director its window belongs to.

The pane initialization method is where you set the pane's location in its enclosure and its characteristics. If you want your pane, or any of the panes it encloses, to receive clicks, be sure to send the pane a `SetWantsClicks(true)` message, otherwise mouse clicks in your pane are ignored.

If your pane allocates memory, you should also override the `Free` method to deallocate it. Be sure that your method calls `inherited Free` to make sure that the pane is disposed of properly.

The `Draw` message tells your pane to draw its contents. You can assume that the port, clip region, and coordinate system have been set up correctly. If your pane uses long coordinates, you need to map from frame coordinates to QuickDraw coordinates before you do any drawing.

When the user clicks in your pane, it gets a `DoClick` message. Your `DoClick` method can either handle the mouse click itself, or it can create a task and send it in a `TrackMouse` message. To learn more about mouse tracking in a pane, see CMouseTask on page 335.

Coordinate Systems in Panes

Panes use two of the coordinate systems in the THINK Class Library: Frame coordinates and QuickDraw Coordinates.

Frame coordinates provide a local coordinate system for a pane. Units in frame coordinates are in pixels, and the point (0, 0) is usually the top, left corner of the pane. If the pane moves within its enclosure, the coordinate system does not change; the top left corner is still (0, 0). The only time this origin point changes is when you scroll the pane. Each pane can choose to use long coordinates or short coordinates.

All drawing and mouse tracking is done in QuickDraw coordinates. This is the coordinate system that the Macintosh Toolbox uses for its drawing operations. QuickDraw coordinates are only valid after a call to `Prepare`.The relationship between QuickDraw coordinates and the other coordinate systems depends on whether a pane is using long frame coordinates or short frame coordinates.

Short coordinates map directly to QuickDraw coordinates. Each element in a short coordinate uses 16-bit values, so a pane that uses short coordinates is limited to the rectangle (–32768, –32768, 32767, 32767). Long coordinates layer a 32-bit coordinate system on top of the QuickDraw 16-bit coordinates. The long coordinate system lets you use a much larger coordinate area for your pane. Since all drawing takes place in QuickDraw coordinates you

have to map the long coordinates to QuickDraw coordinates when you draw in a pane.

If a pane uses long coordinates, the instance variable `fUsingLongCoord`, which CPane inherits from CView is TRUE. Otherwise, `fUsingLongCoord` is FALSE. The default is to not use long coordinates.

The THINK Class Library uses the types `LongRect` and `LongPt` for both long and short corrdinates. You'll notice that most of the descendants of CView that work with points and rectangles use these types.

Note

If you've worked with earlier versions of the THINK Class Library, this is the biggest change you'll notice since it will require some changes to your programs.

These two types are defined like this in `LongCoordinates.h` in C:

```
typedef struct LongPt
{
   long v, h;
} LongPt;

typedef struct LongRect
{
   long top, left, bottom, right;

} LongRect;
```

And like this in `TCL.p` in Pascal:

```
type
   LongPt = record
               case integer of
            1: (
               v, h: longint
               );
            2: (
               vh: array[VHSelect] of longint
               );
            end;
```

```
LongRect = record
             case integer of
           1: (
              top, left,
              bottom, right: longint
              );
           2: (
              topLeft: LongPt;
              botRight: LongPt
              );
           end;
```

If the pane is using short coordinates, frame coordinates and QuickDraw coordiantes are identical, so the values stored in a `LongRect` or in a `LongPt` are in QuickDraw coordinates. To use them with QuickDraw routines, however, you'll need to convert them to the QuickDraw types `Rect` and `Point`. The THINK Class Library provides several utility routines to do these conversions. For a complete listing, see "Long Coordinate Utilities" on page 488.

Drawing in Panes

To draw in a pane use the standard QuickDraw routines. The pane's `Prepare` method sets up the QuickDraw port. If your pane uses short coordinates, you the coordinate system is set up correctly. If your pane uses long coordinates, you need to transform frame coordinates to QuickDraw coordinates before you draw. You can CPane methods `FrameToQD` and `FrameToQDR` convert frame points and rectangles to QuickDraw points and rectangles.

Here's an example. The following `Draw` method draws a line from the top, left of the pane to the bottom right, and then draws a rectangle inset 10 pixels from the edges of the frame.

In C it looks like this:

```
void CCoolPane::Draw (Rect *area)
{
   Rect theRect;

   MoveTo(frame.left, frame.top);
   LineTo(frame.right, frame.bottom);
   LongToQDRect(&frame, &theRect);
   InsetRect(&theRect, 10, 10);
   FrameRect(&theRect);
}
```

In Pascal, the same method looks like this:


```
procedure CCoolPane.Draw (var area: Rect);
   var
      theRect: Rect;
begin
   MoveTo(frame.left, frame.top);
   LineTo(frame.right, frame.bottom);
   LongToQDRect(frame, theRect);
   InsetRect(theRect, 10, 10);
   FrameRect(theRect)
end;
```

Note that to draw the line from one corner to the other, you can use the values directly since they're QuickDraw values. But to perform an operation on a rectangle, you have to convert the `LongRect` into a QuickDraw `Rect`.

If a pane uses long coordinates, all coordinates are in frame coordinates. Before you can do any drawing, you need to map those values to the QuickDraw space on the screen. The CPane methods `FrameToQD` and `FrameToQDR` map frame coordinates into QuickDraw coordinates.

For example, imagine you have a pane that uses long coordinates and you want its `Draw` method to display the text "Hello World." and then draw a rectangle inset 10 pixels from the frame. The location of the string is in an instance variable called `stringLoc` which is declared as a `LongPt`. The location in `stringLoc` could be well outside QuickDraw coordinate space. It might be at (100000, 100000).

This is how you might write the `Draw` method in C:

```
void CMyLongPane::Draw (Rect *area)
{
   Point qdLoc;
   Rect qdRect;

   FrameToQD(&stringLoc, &qdLoc);
   MoveTo(qdLoc.h, qdLoc.v);
   DrawString("\pHello World.");
   FrameToQDR(&frame, &qdRect);
   InsetRect(&qdRect, 10, 10);
   FrameRect(&qdRect);
}
```

In Pascal it would look like this:

```
procedure CMyLongPane.Draw (var area: Rect);
   var
      qdLoc: Point;
      qdRect: Rect;
begin
   FrameToQD(stringLoc, qdLoc);
   MoveTo(qdLoc.h, qdLoc.v);
   DrawString('Hello World.');
   FrameToQDR(frame, qdRect);
   InsetRect(qdRect, 10, 10);
   FrameRect(qdRect)
end;
```

Keep in mind the difference between converting long-coordinate structures to QuickDraw structures and mapping long-coordinate structures to QuickDraw structures. When you're using short coordinates, frame coordinates are the same as QuickDraw coordinates, so all you have to do is translate on data structure to another. When you're using long coordinates, you need to map a portion of long coordinate space into QuickDraw space so you can draw in it.

Variables

Variable	Type	Description
width	integer	Horizontal size in pixels.
height	integer	Vertical size in pixels.
hEncl	longint	Horizontal location in enclosure.
vEncl	longint	Vertical location in enclosure.
hSizing	SizingOption	Horizontal sizing option.
vSizing	SizingOption	Vertical sizing option.
autoRefresh	Boolean	Refresh after a resize?
frame	LongRect	Area for displaying the pane which defines the frame coordinates.
aperture	LongRect	Active drawing area of the pane.
hOrigin	longint	Window left in frame coordinates.
vOrigin	longint	Window top in frame coordinates.
itsEnvironment	CEnvironment	Drawing environment.
printClip	ClipOption	The region to clip to when printing.
printing	Boolean	Is printing in progress?
itsBorder	CPaneBorder	Border of this pane.

Methods

Construction/Destruction methods

IPane

```
procedure IPane (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption);
void IPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialize a pane. Almost all of the descendants of CPane use the same arguments in their initialization methods.

`AnEnclosure` is the enclosing view that contains this pane. Typically the enclosure is either a window or another pane. If your pane is a panorama, its enclosure should be a scroll pane.

`ASupervisor` is the bureaucrat that handles all the commands that this pane won't. Typically, the supervisor is the document (or director) associated with this pane's window.

`AWidth` and `aHeight` are the width and height of the pane in pixels. `AHEncl` and `aVEncl` are the horizontal and vertical position of the pane within its enclosure.

The aHSizing and aVSizing parameters specify what happens to the pane when the size of its enclosure changes. The length and height of a pane changes relative to its original position in the enclosing pane. These are the values that aHSizing can have:

aHSizing value	Meaning
sizFIXEDLEFT	The left edge of the pane is always the same number of pixels from the left edge of the enclosing pane as when it was originally placed.
sizFIXEDRIGHT	The right edge of the pane is always the same number of pixels from the right edge of the enclosing pane as when it was originally placed.
sizFIXEDSTICKY	The left and right edges stick to their original locations in the enclosing pane. If the enclosure scrolls horizontally, the pane will scroll with it.
sizELASTIC	The width of the pane grows and shrinks by the same amount as the enclosing pane.

These are the values the aVSizing can have:

aVSizing value	Meaning
sizFIXEDTOP	The top edge of the pane is always the same number of pixels from the top edge of the enclosing pane as when it was originally placed.
sizFIXEDBOTTOM	The bottom edge of the pane is always the same number of pixels from the bottom edge of the enclosing pane as when it was originally placed.
sizFIXEDSTICKY	The top and bottom edges stick to their original locations in the enclosing pane. If the enclosure scrolls vertically, the pane will scroll with it.
sizELASTIC	The height of the pane grows and shrinks by the same amount as the enclosing pane.

A couple of examples will make this clearer: A vertical scroll bar in a window would be horizontally sizFIXEDRIGHT and vertically sizELASTIC. It has a fixed horizontal length and remains anchored to the right edge of the window. Vertically, it stretches and contracts with the height of the window. A status box in the lower left corner of a window would be sizFIXEDLEFT horizontally and sizFIXEDBOTTOM vertically. It has a constant size and remains anchored to the bottom left corner of the window.

You may want to look at Figure 7-5 on page 109.

IViewRes

```
procedure IViewRes (rType: ResType; resID: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a pane from a resource template. RType is the resource type for the CView subclass you want to initialize. ResID is the resource ID of the resource. AnEnclosure and aSupervisor are the same as for IBorder. This method is inherited from CView.

To initialize a pane from a resource file, use a 'Pane' resource.

IViewTemp

```
procedure IViewTemp (anEnclosure: CView;
   aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of a pane. If you create a subclass of class CPane, be sure your class's Free or Dispose method calls the inherited method as well.

Accessing methods

SetFrameOrigin

```
procedure SetFrameOrigin (fLeft, fTop: longint);
void SetFrameOrigin (long fLeft, long fTop);
```

Set the coordinates of the top left corner of the pane's frame to determine the frame coordinates of the pane.

GetFrame

```
procedure GetFrame (var theFrame: LongRect);
void GetFrame (LongRect *theFrame);
```

Get the frame of the pane. The frame is the rectangle that encloses the pane in pane coordinates. Usually the top and left of this rectangle are 0,0.

GetLengths

```
procedure GetLengths (var theWidth, theHeight:
   integer);
void GetLengths (short *theWidth, short *theHeight);
```

Get the width and the height of the pane.

GetOrigin

```
procedure GetOrigin (var theHOrigin, theVOrigin:
   longint);
void GetOrigin (long *theHOrigin, long *theVOrigin);
```

Get the origin of a pane. The origin of a pane is the top left of the window in frame coordinates. It's the distance from the top left of the window to the (0,0) of the pane. You should not use or override this method.

GetAperture

```
procedure GetAperture (var theAperture: LongRect);
void GetAperture (LongRect *theAperture);
```

Get the aperture of the pane. The aperture is the visible portion of a pane where drawing can occur. The aperture is given in frame coordinates.

Contains

```
function Contains (windPt: Point): Boolean;
Boolean Contains (Point windPt);
```

Returns TRUE if `windPt` is in the aperture of the pane. `WindPt` is given in window coordinates.

ReallyVisible

```
function ReallyVisible: Boolean;
Boolean ReallyVisible (void);
```

Return TRUE if the pane is visible and within QuickDraw space. This method actually returns TRUE if the pane and its enclosure is *potentially* visible. The pane may not actually be on the screen. A pane is on the screen if it's `ReallyVisible` and the aperture isn't empty.

GetPixelExtent

```
procedure GetPixelExtent (var hExtent,
   vExtent: longint);
void GetPixelExtent (long *hExtent, long *vExtent);
```

Get the dimensions of the pane in pixels.

SetPrintClip

```
procedure SetPrintClip (aPrintClip: ClipOption);
void SetPrintClip (ClipOption aPrintClip);
```

Specify the print option to use when printing. `APrintClip` can be one of

aPrintClip value	Meaning
`clipAPERTURE`	Print only what is visible
`clipFRAME`	Print the contents of the frame
`clipPAGE`	Print to fill the page

The default is `clipFRAME`.

GetWindow

```
function GetWindow: CWindow;
CWindow *GetWindow (void);
```

Return the window that encloses the pane. Note that this method returns a window object, not a Macintosh window.

SetBorder

```
procedure SetBorder (aBorder: CPaneBorder);
void SetBorder (CPaneBorder *aBorder)
```

Set the border for this pane. For more information about borders, see the description of CPaneBorder on page 361.

SetResBorder

```
procedure SetResBorder (resID: integer);
void SetResBorder (short resID)
```

Create a border for this pane from a `'PBrd'` resource whose ID is `resID`.

GetBorder

```
function GetBorder: CPaneBorder;
CPaneBorder *GetBorder (void)
```

Return the border for this pane.

GetHelpResID

```
function GetHelpResID: integer;
short GetHelpResID (void)
```

Return the ID of the `'hrct'` resource that has the Balloon Help information for this pane. The resource ID is actually stored in the window object that encloses this pane. For more information about help resources for panes see "Using Balloon Help with views" on page 462.

Appearance methods

Show

```
procedure Show;
void Show (void);
```

Show the pane if it was hidden. The default method sends a `Refresh` message to the pane after making it visible.

Hide

```
procedure Hide;
void Hide (void);
```

Hide the pane if it was visible. The default method sends a `Refresh` message to the pane before hiding it.

Size and location methods

Place

```
procedure Place (hEncl, vEncl: longint;
   redraw: Boolean);
void Place (long hEncl, long vEncl, Boolean redraw);
```

Place the pane at the point `hEncl`, `vEncl` of its enclosure. `HEncl` and `vEncl` are given in the coordinate system of the enclosure. If `redraw` is `TRUE`, redraw the pane after moving it.

Offset

```
procedure Offset (hOffset, vOffset: longint;
   redraw: Boolean);
void Offset (long hOffset, long vOffset,
   Boolean redraw);
```

Offset the pane by `hOffset` pixels horizontally and `vOffset` pixels vertically. If `redraw` is `TRUE`, redraw the pane after moving it.

ChangeSize

```
procedure ChangeSize (delta: Rect; redraw: Boolean);
void ChangeSize (Rect *delta, Boolean redraw);
```

Change the size of the pane. The values of each field of the `delta` rectangle specify how each side should change. Positive values mean down and to the right. Negative values mean up and to the left.

AdjustToEnclosure

```
procedure AdjustToEnclosure (deltaEncl: Rect);
void AdjustToEnclosure (Rect *deltaEncl);
```

Adjust the size or location of the pane when the enclosure has moved or changed size. You should not override or use this method.

AdjustHoriz

```
procedure AdjustHoriz (deltaEncl: Rect;
   var delta: Rect; var offset: integer;
   var moved: Boolean; var sized: Boolean);
void AdjustHoriz (Rect *deltaEncl, Rect *delta,
   short *offset, Boolean *moved, Boolean *sized);
```

Adjust the horizontal size or location of a pane. You should not override or use this method.

AdjustVert

```
procedure AdjustVert (deltaEncl: Rect;
   var delta: Rect; var offset: integer;
   var moved: Boolean; var sized: Boolean);
```



```
void AdjustVert (Rect *deltaEncl, Rect *delta,
   short *offset, Boolean *moved, Boolean *sized);
```

Adjust the vertical size or location of a pane. You should not override or use this method.

EnclosureScrolled

```
procedure EnclosureScrolled (hOffset, vOffset:
   longint);
void EnclosureScrolled (long hOffset, long vOffset);
```

Adjust the location of a pane after its location has scrolled. This method affects the pane only if it is sticky in the direction it is being moved. You should not use or override this method.

Adapting methods

FitToEnclosure

```
procedure FitToEnclosure (horizFit, vertFit: Boolean);
void FitToEnclosure (Boolean horizFit,
   Boolean vertFit);
```

Make the frame of the pane fit the interior of its enclosure in either the vertical or horizontal direction. If `horizFit` is `TRUE`, the left edge and width of the pane's frame change to coincide with the enclosure's interior. If `vertFit` is `TRUE`, the top edge and height of the pane's frame change to coincide with the enclosure's interior. `FitToEnclosure` sends the enclosure a `GetInterior` message to determine the interior of the pane. `FitToEnclosure` does not redraw the pane.

FitToEnclFrame

```
procedure FitToEnclFrame (horizFit, vertFit: Boolean);
void FitToEnclFrame (Boolean horizFit,
   Boolean vertFit);
```

Fit the frame of the pane to the frame of its enclosure in either the vertical or horizontal direction. If `horizFit` is `TRUE`, the left edge and the width of the pane's frame change to coincide with the enclosure's frame. If `vertFit` is `TRUE`, the top edge and the height of the pane's frame change to coincide with the enclosure's frame. `FitToEnclFrame` does not redraw the pane.

CenterWithinEnclosure

```
procedure CenterWithinEnclosure (horizCenter,
   vertCenter: Boolean);
void CenterWithinEnclosure (Boolean horizCenter,
   Boolean vertCenter);
```

Center the pane within its enclosure horizontally or vertically. Only the location of the pane changes. The size of the pane does not change. `CenterWithinEnclosure` does not redraw the pane.

Drawing methods

Draw

```
procedure Draw (area: Rect);
void Draw (Rect *area);
```

Draw the contents of the pane. The `area` parameter specifies the portion of the pane that needs to be redrawn. Your subclass must override this method.

If the pane is not using long coordinates, the `area` parameter is given in frame coordinates, which in this case are the same as QuickDraw coordinates.

If the pane uses long coordinates, area is given in QuickDraw coordinates. Your `Draw` method can use `QDToFrame` or `QDToFrameR` to convert from QuickDraw coordinates to frame coordinates and `FrameToQD` and `FrameToQDR` to convert from frame coordinates to QuickDraw coordinates.

DrawAll

```
procedure DrawAll (var area: Rect);
void DrawAll (Rect *area);
```

Draw the pane and all its subviews and any borders that the pane or subviews may have. Use this method when you want to force the entire pane to be redrawn without waiting for an update event. Scrolling is a good example. You want to redraw the pane as soon as it has scrolled instead of waiting for an update event. This method prepares the pane before sending `Draw` messages to it and to its subpanes. You should not override this method. Look at the implementation of CPanorama's `Scroll` method for an example of using `DrawAll`.

Refresh

```
procedure Refresh;
void Refresh (void);
```

Force the pane to redraw itself on the next update event. Default method sends a `RefreshLongRect` message to the pane using the frame as the area.

RefreshRect

```
procedure RefreshRect (area: Rect);
void RefreshRect (Rect *area);
```

Force a portion of the pane to redraw itself on the next update event. `Area` is a `Rect` in frame coordinates. Only the portion of the pane that's actually visible will actually be redrawn.

RefreshLongRect

```
procedure RefreshLongRect (area: LongRect);
void RefreshLongRect (LongRect *area);
```

Force a portion of the pane to redraw itself on the next update event. `Area` is a `LongRect` in frame coordinates. Only the portion of the pane that's actually visible will actually be redrawn.

RefreshBorder

```
procedure RefreshBorder;
void RefreshBorder (void);
```

Force the pane's border to redraw itself on the next update event.

Printing methods

Paginate

```
procedure Paginate (aPrinter: CPrinter;
   pageWidth, pageHeight: integer);
void Paginate (CPrinter *aPrinter, short pageWidth,
   short pageHeight);
```

Set up the pane's pagination. By default, panes print only on one page. The document that a pane belongs to sends this message to the pane. For examples more complex pagination, see CAbstractText and CPanorama.

AboutToPrint

```
procedure AboutToPrint (var firstPage, lastPage:
   integer);
void AboutToPrint (short *firstPage, short *lastPage);
```

Printing is about to begin. Your pane subclass can override this method to do anything that needs to happen right before printing.

PrintPage

```
procedure PrintPage (pageNum, pageWidth, pageHeight:
   integer; aPrinter: CPrinter);
void PrintPage (short pageNum, short pageWidth,
   short pageHeight, CPrinter *aPrinter);
```

Print the specified page of the pane. By default, panes have only on page. `APrinter` is usually supplied by the document that this pane belongs to. If your pane subclass supports multi-page documents, you must override this method to determine which part of the pane to draw.

DonePrinting

```
procedure DonePrinting;
void DonePrinting (void);
```

Printing is over. If you want to take any action when printing is done, override this method.

PrepareToPrint

```
procedure PrepareToPrint;
void PrepareToPrint (void);
```

Set up the coordinate system and the clipping region before printing. The default method sets the clipping region to the region described in the `printClip` variable. You can set this variable with the `SetPrintClip` method.

Calibration methods

Prepare

```
procedure Prepare;
void Prepare (void);
```

Prepare the pane for drawing. `Prepare` sets up the port and the QuickDraw coordinates for the pane, converting lon coordinates to QuickDraw coordinates if necessary. It also sets the clipping region to the aperture (the visible portion of the pane) so drawing is constrained to the visible portion of the pane. If the pane has an environment associated with it, `Prepare` sends a `Restore` message to the environment. If the pane is being printed, the default method sends the pane a `PrepareToPrint` message instead.

The inherited `Prepare` method in CView sets `cPreparedView` to this view. If this view gets another `Prepare` message, `Prepare` avoids setting up the drawing environment all over again.

RestoreEnvironment

```
procedure RestoreEnvironment;
void RestoreEnvironment (void);
```

Restore the pane's drawing environment. If the pane has a drawing environment associated with it, this method sends a `Restore` message to it.

CalcFrame

```
procedure CalcFrame;
void CalcFrame (void);
```

Calculate the coordinates of the pane's frame based on the frame's width and height and its location within its enclosure. Generally, the superclasses of the pane classes you define will send this message. You should not use or override this method unless your pane subclass needs something other than (0,0) at its top left corner

ResizeFrame

```
procedure ResizeFrame (delta: Rect);
void ResizeFrame (Rect *delta);
```

Adjust the frame when the size of the pane changes. The `delta` rectangle specifies how each side changes. Positive values mean down and to the right. Negative values mean up and to the left. The default method always sets the top left of the frame to (0,0)

You should not need to use or override this method unless your pane subclass requires that the top, left of the frame be something other than (0,0). See the implementation of `ResizeFrame` in CPanorama for an example.

CalcAperture

```
procedure CalcAperture;
void CalcAperture (void);
```

Calculate the visible (or drawable) portion of a pane. The aperture of a pane is the area that is not obscured by the bounds of the enclosing view. You should not use or override this method. To get the aperture, use `GetAperture` on page 350.

Cursor method

TrackMouse

```
procedure TrackMouse (theTask: CMouseTask;
   startPt: LongPt; var pinRect: LongRect);
void TrackMouse (CMouseTask *theTask, LongPt startPt,
   LongRect *pinRect);
```

To learn more about mouse trackingand mouse tasks, see the class CMouseTask on page 335.

Track the mouse in this view. `TheTask` is a mouse task that you create. `StartPt` is the starting point in frame coordinates. `PinRect` is a constraining rectangle in frame coordinates. This message is usually sent from a `DoClick` method.

The default method does several things:

- Sends a `Prepare` message to the view
- Sends a `BeginTracking` message to `theTask`
- Sends a `KeepTracking` message to `theTask` as long as the mouse is down. The current point is constrained to `pinRect`. (Your task subclass must override the `KeepTracking` method.)
- Sends an `EndTracking` message to `theTask`. (Your task subclass must override this method as well.)

Coordinate transformation methods

WindToFrame

```
procedure WindToFrame (windPt: Point;
   var framePt: LongPt);
void WindToFrame (Point windPt, LongPt *framePt);
```

Convert the point `windPt` from window coordinates to frame coordinates and place the converted point in `framePt`.

WindToFrameR

```
procedure WindToFrameR (windRect: Rect,
   var frameRect: LongRect);
void WindToFrameR (Rect *windRect,
   LongRect *frameRect);
```

Convert the rectangle `windRect` from window coordinates to frame coordinates and place the converted rectangle in `frameRect`.

FrameToWind

```
procedure FrameToWind (framePt: LongPt;
   var windPt: Point);
void FrameToWind (LongPt *framePt, Point *windPt);
```

Convert the point `framePt` from frame coordinates to window coordinates and place the converted point in `windPt`.

FrameToWindR

```
procedure FrameToWindR (frameRect: LongRect;
   var windRect: Rect);
void FrameToWindR (LongRect *frameRect,
   Rect *windRect);
```

Convert the rectangle `frameRect` from frame coordinates to window coordinates and place the converted rectangle in `windRect`.

EnclToFrame

```
procedure EnclToFrame (var thePoint: LongPt);
void EnclToFrame (LongPt *thePoint);
```

Convert a point from the frame coordinates of its enclosure to the frame coordinates of the pane.

EnclToFrameR

```
procedure EnclToFrameR (var theRect: LongRect);
void EnclToFrameR (LongRect *theRect);
```

Convert a rectangle from the frame coordinates of its enclosure to the frame coordinates of the pane.

FrameToEncl

```
procedure FrameToEncl (var thePoint: LongPt);
void FrameToEncl (LongPt *thePoint);
```

Convert a point from frame coordinates to the frame coordinates of its enclosure.

FrameToEnclR

```
procedure FrameToEnclR (var theRect: LongRect);
void FrameToEnclR (LongRect *theRect);
```

Convert a rectangle from frame coordinates to the frame coordinates of its enclosure.

FrameToGlobalR

```
procedure FrameToGlobalR (frameRect: LongRect;
   globalRect: Rect);
void FrameToGlobalR (LongRect *frameRect,
   Rect *globalRect);
```

Convert the rectangle `frameRect` from frame coordinates to global coordinates and place the converted rectangle in `globalRect`.

QDToFrame

```
procedure QDToFrame (qdPoint: Point;
   var framePt: LongPt);
void QDToFrame (Point qdPoint, LongPt *framePt);
```

Convert `qdPoint` from QuickDraw coordinates to frame coordinates, and put the result in `framePt`. `QdPoint` is assumed to be in the portion of QuickDraw space that the pane is mapping to. If the pane is not using long coordinates, the values in `qdPoint` and `framePoint` are the same.

QDToFrameR

```
procedure QDToFrameR (qdRect: Rect;
   var frameRect: LongRect);
void QDToFrameR (Rect *qdRect, LongRect *frameRect);
```

Convert `qdRect` from QuickDraw coordinates to frame coordinates, and put the result in `frameRect`.

FrameToQD

```
procedure FrameToQD (framePt: LongPt;
   var qdPt: Point);
void FrameToQD (LongPt *framePt, *Point qdPt);
```

Convert `framePt` from frame coordinates to QuickDraw coordinates, and put the results in `qdPoint`.

FrameToQDR

```
procedure FrameToQDR (frameRect: LongRect;
   var qdRect: Rect);
void FrameToQDR (LongRect *frameRect, Rect *qdRect);
```

Convert `frameRect` from frame coordinates to QuickDraw coorinates, and put the result in `qdRect`.

SectAperture

```
function SectAperture (srcRect: LongRect;
   var destRect: Rect): Boolean;
Boolean SectAperture (LongRect *srcRect,
   Rect *destRect);
```

Clip the `srcRect` (in frame coorindate) to the pixels visible through the aperture, and return the result in a QuickDraw rectangle, `destRect`. If `destRect` is not empty, this method returns TRUE.

CPaneBorder 44

Introduction

CPaneBorder is a class that draws a border around a pane.

Heritage

Superclass	CObject
Subclasses	None

Using CPaneBorder

CPaneBorder draws a border around a pane. You can choose from several types, including a rectangle, a rectangle with a shadow, and a rounded rectangle.If you need a different type, create a subclass of CPaneBorder. This class doesn't use any coordinate system. Its parameters are described as offsets from the pane's frame.

To give a pane a border, create an instance of CPaneBorder and send the pane a `SetBorder` message, with your border as the argument. The pane takes care of drawing it. When the size of the pane changes, the size of the border changes automatically. If you want to access the border to change it, send the pane a `GetBorder` message.

When you initialize the border with `IPaneBorder`, you set its shape. These are the available shapes:

Name	Description
`kBorderFrame`	Rectangle
`kBorderRoundRect`	Rounded rectangle
`kBorderOval`	Oval
`kBorderLeft`	Line on left side
`kBorderTop`	Line on top side
`kBorderRight`	Line on right side
`kBorderBottom`	Line on bottom
`kBorderNone`	No border

You can add together `kBorderLeft`, `kBorderTop`, `kBorderRight`, and `kBroderBottom` to create new shapes. For example, `kBorderTop + kBorderRight` gives you a border with lines on the top and right of the pane. To create yet more shapes, you need to subclass CPaneBorder.

After you initialize your border, you can change its attributes, like the width of the border outline, the distance between the border outline and the frame, and whether there's a shadow. Figure 44-1 shows you the instance variables that control these attributes:

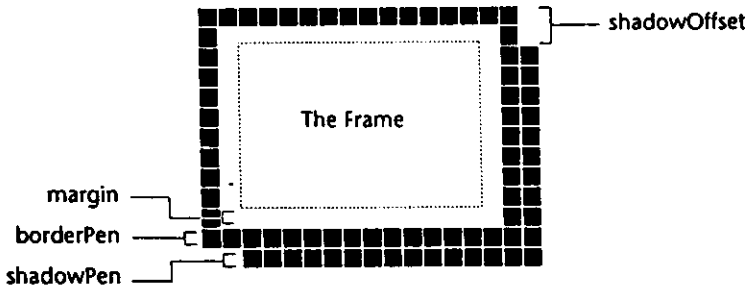

Figure 44-1 A pane border

This table describes what these instance variables control and the methods to set them:

Variable	Method	Description
`margin`	`SetMargin`	Distance between the frame and the border.
`borderPen`	`SetPenSize`	Width of the pen used to draw the border.
`doShadow`	`SetShadow`	TRUE, if this border has a shadow.
`shadowPen`	`SetShadow`	Width of the pen used to draw the shadow.
`shadowOffset`	`SetShadow`	Distance between the border and its shadow.

This table describes other instance variables that control the border's appearance:

Variable	Method	Description
`borderFlags`	`SetBorderFlags`	The shape of the border.
`penpat`	`SetPattern`	The pattern used to draw the border and shadow.
`roundDiameter`	`SetRounding`	The roundness of the border's corners, if the shape is a rounded rectangle.

Variables

These are internal instance variables. You can access them with the accessing methods described below. Only subclasses of CPaneBorder should refer to them directly. In THINK C, the instance variables are protected.

Variable	Type	Description
`borderFlags`	`long`	The shape of the border.
`borderPen`	`Point`	The width of the pen used to draw the border.
`shadowOffset`	`Point`	The distance between the border and its shadow.
`shadowPen`	`Point`	The width of the pen used to draw the shadow.
`doShadow`	`Boolean`	TRUE if shadow is drawn.
`roundDiameter`	`Point`	The roundness of the border's corners, if the shape is a rounded rectangle.
`penPat`	`Pattern`	The pattern used to draw the border and shadow..
`margin`	`Rect`	The distance between the frame and the border.

Methods

Creation and destruction

IPaneBorder

```
procedure IPaneBorder (borderFlags: integer);
void IPaneBorder (short borderFlags);
```

Initialize a pane border. The `borderFlags` describe the shape of the border. It can be one of these :`kBorderNone`, `kBorderLeft`, `kBorderTop`, `kBorderRight`, `kBorderBottom`, `kBorderOval`, `kBorderRoundRect`, `kBorderFrame`. All the defaults may be changed via methods.

IResPaneBorder

```
procedure IResPaneBorder (resID: integer);
void IResPaneBorder (short resID);
```

Initialize a border from a `'PBrd'` resource.

Accessing methods

SetPattern

```
procedure SetPattern (aPattern: Pattern);
void SetPattern (Pattern aPattern);
```

Set the pattern used for drawing the border's outline and its shadow.

GetPattern

```
procedure GetPattern (var aPattern: Pattern);
void GetPattern (Pattern aPattern);
```

Return the current pattern.

SetBorderFlags

```
procedure SetBorderFlags (aBorderFlags: integer);
void SetBorderFlags (short aBorderFlags);
```

Set the border flags. `ABorderFlags` can be one of these: `kBorderNone`, `kBorderLeft`, `kBorderTop`, `kBorderRight`, `kBorderBottom`, `kBorderOval`, `kBorderRoundRect`, `kBorderFrame`.

GetBorderFlags

```
function GetBorderFlags: longint;
long GetBorderFlags (void);
```

Return the current border flags.

SetPenSize

```
procedure SetPenSize (penWidth, penHeight: integer);
void SetPenSize (short penWidth, short penHeight);
```

Set the pen size used to draw the border.

GetPenSize

```
procedure GetPenSize (var penWidth, penHeight:
   integer);
void GetPenSize (short *penWidth; short *penHeight);
```

Return the pen size used to draw the border.

SetShadow

```
procedure SetShadow (hOffset, vOffset,
   width, height: integer);
void SetShadow (short hOffset, short vOffset,
   short width, short height);
```

Set the shadow attributes. `HOffset` and `vOffset` specify the distance of the shadow from the border outline. `Width` and `height` specify the pen size used to draw the shadow. This method also sets `doShadow` to TRUE.

GetShadow

```
function GetShadow (var hOffset, vOffset,
   width, height: integer): longint;
long GetPenSize (short *hOffset, short *vOffset,
   short *width, short *height);
```

Return the current shadow attributes.

SetRounding

```
procedure SetRounding (hDiameter, vDiameter: integer);
void SetRounding (short hDiameter, short vDiameter);
```

Set the roundness of the corners of rounded rectangles. These are used as parameters to `FrameRoundRect`. This method also sets the border style to `kBorderRoundRect`, if that isn't the border style already.

GetRounding

```
procedure GetRounding (var hDiameter, vDiameter:
   integer);
void GetRounding (short *hDiameter, short *vDiameter);
```

Returns the current rounding diameters.

SetMargin

```
procedure SetMargin (aMargin: Rect);
void SetMargin (Rect *aMargin);
```

Set the distance between the border outline and the frame. `AMargin` is a `Rect` with positive offsets for each side of the border. For example, to have a one pixel margin on all sides, use (1, 1, 1, 1).

GetMargin

```
procedure GetMargin (var aMargin: Rect);
void GetMargin (Margin *aMargin);
```

Return the current border margin.

Drawing method

DrawBorder

```
procedure DrawBorder (paneFrame: Rect);
void DrawBorder (Rect *paneFrame);
```

Draw the border. `PaneFrame` is the pane's frame in the pane's current coordinates.

Calibration method

CalcBorderRect

```
procedure CalcBorderRect (var paneFrame: Rect);
void CalcBorderRect (Rect *paneFrame);
```

Return the area for the frame and its border. `PaneFrame` is the pane's frame in the pane's current coordinates. This method adds to it the size of the border, including the margin and shadow, on all sides.

CPaneMDEF 45

Introduction

CPaneMDEF is an abstract class that lets you display any pane as a Macintosh Menu. The class includes methods for handling tear-off menus.

Heritage

Superclass	CMenuDefProc
Subclasses	CSelectorMDEF

Using CPaneMDEF

CTearOffMenu is described on page 443.

CPaneMDEF's superclass, CMenuDefProc, lets you write object-oriented menu definition procedures. This class and its descendants take that idea a step further to let you use THINK Class Library panes as menus. The pane that you use in a CPaneMDEF subclass should belong to a subclass of CTearOffMenu.

A tear-off menu is a window that appears to float above all the other windows in your application. In that environment, the pane is part of the visual hierarchy, and it will behave just like any other pane in a window.

When it appears as a custom menu, the pane isn't really in the visual hierarchy at all. As far as the pane is concerned, its enclosure is the floating window it appears in when it's torn off. When you draw a custom menu in your `Draw` method , you have to set up the QuickDraw environment by hand with the `SetupQuickDraw` method. This method changes the QuickDraw coordinate system so drawing takes place in pane coordinates. Be sure to call `RestoreQuickDraw` at the end of the `Draw` method

Since this class doesn't have a `ChooseItem` method, you'll need to create a subclass of CPaneMDEF that handles item selection. The class CSelectorMDEF is a subclass of CPaneMDEF that lets you use selectors as menus. It's also a good class to study to learn how to write a `ChooseItem`

method. Your ChooseItem method should also use the SetupQuickDraw/RestoreQuickDraw methods.

Variables

Variable	Type	Description
itsPane	CPane	The pane to display as a menu.
itsTearOffMenu	CTearOffMenu	The torn-off menu.
savePort	GrafPtr	Used internally to save the grafport.
saveClip	RgnHandle	Used internally to save the clipping region.

Methods

Construction and destruction methods

IPaneMDEF

```
procedure IPaneMDEF (MDEFid: integer;
   aPane: CPane; aTearOffMenu: CTearOffMenu);
void IPaneMDEF (short MDEFid, CPane *aPane,
   CTearOffMenu *aTearOffMenu);
```

Initialize the pane MDEF. This method calls IMenuDefProc with the MDEFid. The instance variable itsPane is set to aPane, which is the pane that you want to display as a menu. ATearOffMenu is a tear-off menu object. You can pass NIL if this menu is not a tear-off menu. See the CTearOffMenu class on page 443.

DrawMenu

```
procedure DrawMenu (macMenu: MenuHandle;
   menuRect: Rect);
void DrawMenu (MenuHandle macMenu, Rect *menuRect);
```

This method draws the pane as a menu. MacMenu and menuRect are provided by the Macintosh Menu Manager and are in global coordinates. DrawMenu uses the SetupQuickDraw method to set up the QuickDraw drawing environment so the point (0,0) is the top, left corner of the pane. The method then sends itsPane a RestoreEnvironment message and then a Draw message to display the pane. If the menu is disabled, DrawMenu grays out the menu. Finally, the method restores the QuickDraw environment.

Unless your subclass needs to do something extraordinary to draw your menu, you should not override this method.

SizeMenu

```
procedure SizeMenu (macMenu: MenuHandle);
void SizeMenu (MenuHandle macMenu);
```

This method is called by the Macintosh Menu Manager to determine the size of the menu. This method gets the height and width of the pane and stores them in menuWidth and menuHeight fields of macMenu. Your subclass should not override this method.

TearOffMenu

```
procedure TearOffMenu (menuRect: Rect;
   mouseLoc: Point);
void TearOffMenu (Rect *menuRect, Point mouseLoc);
```

TearOffMenu drags a gray outline representing the menu when you tear it off the menu bar. The size of the gray outline is the menuRect plus the margins specified in itsTearOffMenu. You can drag the outline as long as the mouse is still down, and the mouse is in the tear-off region. If the mouse is in the tear-off region when the button is released, TearOffMenu sends a TornOff message to itsTearOffMenu to do the "tearing."

This method doesn't change the QuickDraw coordinates because it expects to be called from ChooseItem *which has already set things up.*

Unless your subclass needs to do something fancy to show that a menu is being torn off, you should not override this method.

Your subclass should call this method in its ChooseItem method when the hitPt is not in the menuRect. You should also check to make sure that there is a tear-off menu (itsTearOffMenu is not NIL) and that the menu is not disabled.

For a good example of a ChooseItem *method, see the CSelector class.*

Here's what this might look like in Pascal:

```
procedure CMyMDEF.ChooseItem(macMenu: MenuHandle
   menuRect: Rect; hitPt: Point;
   var whichItem: integer);
begin
   if PtInRect(hitPt, menuRect) then
      begin
      SetupQuickDraw(menuRect);
         { choose an item }
      RestoreQuickDraw;
      end
   else
      begin
         whichItem = NOTHING;
         if (itsTearOffMenu <> NIL) AND
            MenuEnabled(macMenu)
         then
            TearOffMenu(menuRect, hitPt)
      end
end
```

In C it looks like this:

```
void CMyMDEF::ChooseItem(MenuHandle macMenu,
    Rect *menuRect, Point hitPt,
   short *whichItem)
{
   if (PtInRect(hitPt, menuRect) {
      SetupQuickDraw(menuRect);
      /* choose an item */
      RestoreQuickDraw();
   } else {
      *whichItem = NOTHING;
      if (itsTearOffMenu != NULL &&
            MenuEnabled(macMenu))
         TearOffMenu(menuRect, hitPt);
   }
}
```

PtInTearRgn

```
function PtInTearRgn (hitPt: Point;
   menuRect: Rect): Boolean;
Boolean PtInTearRgn (Point hitPt, Rect *menuRect);
```

Returns TRUE if `hitPt` (given in global coordinates) is in the tear-off region. The tear-off region is anywhere except the menu bar and within `TEARMARGIN` pixels of the menu.

SetupQuickDraw

```
procedure SetupQuickDraw (menuRect: Rect);
void SetupQuickDraw (Rect *menuRect);
```

Set up the QuickDraw environment so the MDEF's pane draws correctly. Since panes associated with MDEFs aren't really part of the visual hierarchy, you have to set up the QuickDraw environment directly. This method sets the port to the desktop's port and sets the origin to the top left of the pane and sets the clipping region to `menuRect`.

RestoreQuickDraw

```
procedure RestoreQuickDraw;
void RestoreQuickDraw (void);
```

Restore the QuickDraw environment to what it was before the `SetupQuickDraw` call.

CPanorama 46

Introduction

CPanorama is an abstract class for implementing displays that may be larger than the frame of a pane. The frame is a viewport though which a portion of the panorama is visible.

Heritage

Superclass	CPane
Subclasses	CPicture
	CStaticText

Using CPanorama

Use a subclass of CPanorama whenever you want to display something that is larger than the pane you want to view it through. For example, some pictures are much bigger than a standard Macintosh screen. For an example of a panorama, look at the CEditText class on page 295. In almost every case, you'll use a panorama with a scroll pane (see CScrollPane on page 413 for a description).

Drawing in a panorama is almost the same as drawing in a pane. The main difference is that panoramas can have their own coordinate system. Each panorama unit can map to one or more pixels. For example, in the CEditText class, the horizontal unit is set to the number of pixels in the widest character, and the vertical unit is set to the number of pixels of the height of a line. Scroll panes use this information to set the scroll bars accurately.

Note

Panoramas do not support fractional scales or non-linear scales for coordinate systems.

The size of the panorama image is called the **bounds**. In most cases, the top, left corner of the bounds is the point (0, 0), but it's not required. The way to

think of the relationship between the panorama and the frame of the pane is that the panorama image is stationary as the frame roves over it. As you move around in a panorama, the coordinates of the top, left corner of the frame will change.

Variables

Variable	Type	Description
bounds	LongRect	Bounds defining Pane coordinates.
position	LongPoint	Location of frame in panorama.
hScale	integer	Pixels per horizontal unit.
vScale	integer	Pixels per vertical unit.
savePosition	LongPoint	Save for later restoration.
itsScrollPane	CScrollPane	Scroll pane a panorama belongs to, if any.

Methods

Construction and destruction methods

IPanorama

```
procedure IPanorama (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption);
void IPanorama (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialize a panorama. The arguments to this routine are identical to the ones for IPane.

Note

The descriptions of the other arguments are in CPane on page 347.

IViewRes

```
procedure IViewRes (rType: ResType; resID: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a panorama from a resource template. Rtype is the resource type for the CView subclass you want to initialize. ResID is the resource ID of the resource. AnEnclosure and aSupervisor are the same as for IBorder. This method is inherited from CView.

To initialize a panorama from a resource file, use a 'Pano' resource.

IViewTemp

```
procedure IViewTemp (anEnclosure: CView;
   aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

Accessing methods

GetExtent

```
procedure GetExtent (var theHExtent, theVExtent:
   longint);
void GetExtent (long *theHExtent, long *theVExtent);
```

Get the size of each side of the panorama in panorama units.

GetFramePosition

```
procedure GetFramePosition (var theHPos, theVPos:
   longint);
void GetFramePosition (long *theHPos, long *theVPos);
```

Determine how far the top, left of the frame is from the top, left of the bounds in panorama coordinates. The CScrollPane class uses this method to figure out the maximum settings for the scroll bars. You should not use or override this method.

GetFrameSpan

```
procedure GetFrameSpan (var theHSpan, theVSpan:
   integer);
void GetFrameSpan (short *theHSpan, short *theVSpan);
```

Return the number of panorama units that the frame spans.

SetBounds

```
procedure SetBounds (aBounds: LongRect);
void SetBounds (LongRect *aBounds);
```

Set the bounds of the panorama. The bounds define the size of the data displayed in the panorama and the panorama coordinates. If the panorama's

enclosure is a scroll pane, this method sends it an `AdjustScrollMax` message to adjust the scroll bars.

GetBounds

```
procedure GetBounds (var theBounds: LongRect);
void GetBounds (LongRect *theBounds);
```

Get the bounds of a panorama.

SetPosition

```
procedure SetPosition (aPosition: LongPt);
void SetPosition (LongPt aPosition);
```

Set the position of the frame in relation to the panorama. The point is in panorama coordinates. If the panorama's enclosure is a scroll pane, this method sends it a `Calibrate` message to adjust the position of scroll box (the scroll bars' "thumb").

GetPosition

```
procedure GetPosition (var thePosition: LongPt);
void GetPosition (LongPt *thePosition);
```

Get the position of the frame in relation to the panorama. In other words, what are the panorama coordinates of the top, left of the frame?

SetScales

```
procedure SetScales (aHScale, aVScale: integer);
void SetScales (short aHScale, short aVScale);
```

Set the horizontal and vertical scales. Specify the scales in pixels per panorama unit. For instance, the CStaticText class (a subclass of CPanorama) uses the width of the widest character as its horizontal unit and the height of a line as a vertical unit. If the panorama's enclosure is a scroll pane, this method sends it an `AdjustScrollMax` message to adjust the scroll bars.

GetScales

```
procedure GetScales (var theHScale, theVScale:
   integer);
void GetScales (short *theHScale, short *theVScale);
```

Get the scale factors of the panorama.

SetScrollPane

```
procedure SetScrollPane (aScrollPane: CScrollPane);
void SetScrollPane (CScrollPane *aScrollPane);
```

Specify the scroll pane that controls this panorama. This method is for use only by the CPanorama class. To associate a panorama with a scroll pane, use the `InstallPanorama` method.

GetHomePosition

```
procedure GetHomePosition (var theHomePos: LongPt);
void GetHomePosition (LongPt *theHomePos);
```

Get the location of the top, left corner of the panorama in panorama coordinates.

GetPixelExtent

```
procedure GetPixelExtent (var hExtent, vExtent:
   longint);
void GetPixelExtent (long *hExtent, long *vExtent);
```

Get the size of each side of the panorama in pixels.

Calibrating methods

ResizeFrame

```
procedure ResizeFrame (delta: Rect);
void ResizeFrame (Rect *delta);
```

Adjust the frame of a panorama when its size changes. The `delta` rectangle specifies the amount of change for each side. Positive numbers mean down and to the right. Negative numbers mean up and to the left.

Scrolling methods

Scroll

```
procedure Scroll (hDelta, vDelta: longint;
   redraw: Boolean);
void Scroll (long hDelta, long vDelta,
   Boolean redraw);
```

Scroll a panorama by `hDelta` units horizontally and `vDelta` units vertically. The units are given in panorama coordinates.

ScrollTo

```
procedure ScrollTo (aPosition: LongPt;
   redraw: Boolean);
void ScrollTo (PointLongPt, Boolean redraw);
```

Scroll the panorama to a specific position. The units given in `aPosition` are in panorama coordinates.

ScrollToSelection

```
procedure ScrollToSelection;
void ScrollToSelection (void);
```

Scroll the panorama so the selection is visible. The default method does nothing. Your panorama subclass must override this method.

AutoScroll

```
function AutoScroll (mouseLoc: Point): Boolean;
Boolean AutoScroll (Point mouseLoc);
```

Scroll automatically during a mouse-down. Returns TRUE if scrolling actually took place. Typically, you would send this message in the same routine that tracks selection.

DoKeyDown

```
procedure DoKeyDown (theChar: char; keyCode: Byte;
   macEvent: EventRecord)
void DoKeyDown (char theChar, Byte keyCode,
   EventRecord *macEvent);
```

This method supports the Home, End, Page Up, and Page Down keys of the extended keyboard. The Home and End keys send `ScrollTo` messages to the panorama's scroll pane to scroll to the beginning or end of the panorama. The Page Up and Page Down keys send `DoVertScroll` messages to the panorama's scroll pane to simulate hits on the page up and page down regions of the scroll bar.

Printing methods

Paginate

```
procedure Paginate (aPrinter: CPrinter;
   pageWidth, pageHeight: integer);
void Paginate (CPrinter *aPrinter, short pageWidth,
   short pageHeight);
```

Determine how many pages to print. The panorama is divided into horizontal and vertical strips of equal size. `APrinter` is the printing object. Typically it belongs to the document that owns this panorma.

AboutToPrint

```
procedure AboutToPrint (var firstPage, lastPage:
   integer);
void AboutToPrint (short *firstPage, short *lastPage);
```

The specified range of pages are about to be printed. You can override this method if your subclass needs to take some action before printing.

PrintPage

```
procedure PrintPage (pageNum, pageWidth,
   pageHeight: integer; aPrinter: CPrinter);
void PrintPage(short pageNum, short pageWidth,
   short pageHeight, CPrinter *aPrinter);
```

Print the specified page. `PageWidth` is the width of the page in pixels. `PageHeight` is the height of the page in pixels. `APrinter` is the printer object. Typically it belongs to the document that owns this panorama.

DonePrinting

```
procedure DonePrinting;
void DonePrinting (void);
```

Printing has stopped. Override this method if you need to do some cleanup after printing.

CPatternGrid 47

Introduction

CPatternGrid is a subclass of CGridSelector that displays patterns in a table and lets you choose one. CPatternGrid is useful for implementing pattern palettes like MacPaint and HyperCard.

Heritage

Superclass	CGridSelector
Subclasses	None

Using CPatternGrid

The CPatternGrid class lets you create panes that display patterns in a table. The most common use for this kind of table is a pattern palette for a painting or drawing program. You can use a CPatternGrid as a tool palette that's part of a window or as a custom tear-off menu. The Art Class demonstation program uses CPatternGrid to display it's **Patterns** tear-off menu.

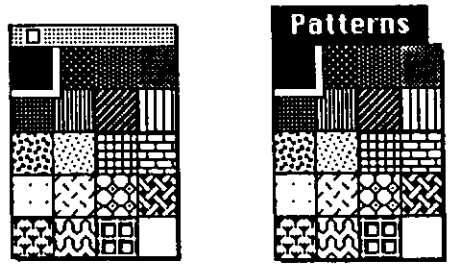

Figure 47-1 Art Class uses CPatternGrid as tear-off menu.

Unlike other subclasses of CPane that optionally let you initialize an object from a resource, you *must* use a resource to initialize a CPatternGrid. The first argument to the initialization method, `IPatternGrid`, is the resource ID of a `'PtGd'` resource which contains the values that `IPatternGrid`

passes up to CGridSelector's initialization method. The 'PtGd' resource looks like this:

Field	Size	Description
Rows	integer	The number of rows in the grid.
Columns	integer	The number of columns in the gird.
Box Width	integer	The width in pixels of each box.
Box Height	integer	The height in pixels of each box.
Horiz. Sizing	integer	Horizontal sizing option. Usually sizFIXEDLEFT (0)
Vert. Sizing	integer	Vertical sizing option. Usually sizFIXEDTOP (2)
Horiz. Location	integer	Horizontal location of grid in its enclosure.
Vert. Location	integer	Vertical location of grid in its enclosure.
Command base	integer	The command base for turning selections into command numbers.
Pattern List ID	integer	The resource ID of the pattern list resource ('PAT#') to use.
Number of patterns	integer	The number of patterns to display.
Pattern indices	integer	The index in the PAT# resource to display. One entry for each pattern.

The standard pattern list is in the System file. Its resource ID is zero.

The first nine fields are the same as the arguments that you pass to IGridSelector. The last three are specific to pattern grids. The Pattern List ID specifies which pattern list resource ('PAT#') the pattern grid should use. The Number of patterns field is how many of the patterns in the list to use. The Pattern indices are the indices of a pattern in the pattern list.

The file TCL TMPLs contains ResEdit templates (TMPL resources) that help you create and edit PtGd resources. This is what the 'PtGd' resource in Art Class looks like when you edit it with ResEdit.

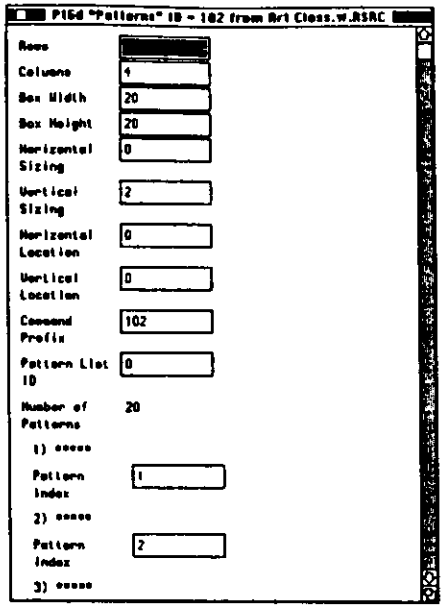

Figure 47-2 The PtGd resource in Art Class

Variables

Variable	Type	Description
patListID	integer	The ID of the 'PAT#' resource to use.
numPats	integer	Number of patterns to use
thePatterns	Handle	Handle to the list of 'PAT#' resource indices.

Methods

Construction methods

IPatternGrid

```
procedure IPatternGrid (PtGdid: integer;
   anEnclosure: CView. aSupervisor: CBureaucrat);
```

```
void IPatternGrid (short PtGdid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialize the character grid. PtGdid is the resource ID of the 'PtGd' resource that describes the location of the pane and pattern list to use for the grid. See "Using CPatternGrid" above for the details on 'PtGd' resources.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the index list and the object.

Drawing methods

DrawItem

```
procedure DrawItem (theItem: integer;
   theBox: Rect);
void DrawItem (short theItem, Rect *theBox);
```

Draw pattern numbered `theItem` so it fills the box. Remember that the `theItem` doesn't specify the `theItem`'th pattern in the pattern list, but the `theItem`'th index in the index list. Patterns are numbered from 1.

HiliteItem

```
procedure HiliteItem (theItem: integer;
   state: HiliteState);
void HiliteItem (short theItem, HiliteState state);
```

Hilight the specified item. If state is `hitliteON`, this method draws a white outline around the box. If state is `hiliteOFF`, it draws the box normally. When state is `hiliteDYNAMIC`, `HiliteItem` flashes the edges of the box. Figure 47-1 shows how CPatternGrid highlights items.

Accessing methods

GetPattern

```
procedure GetPattern (var thePattern: Pattern);
void GetPattern (Pattern *thePattern);
```

Return the currently selected pattern in `thePattern`. The current selection is stored in the `selection` instance variable of the CSelector superclass.

CPictFile 48

Introduction

CPictFile is a class for reading and writing "draw" files like the kind MacDraw produces.

Heritage

Superclass	CDataFile
Subclasses	None

Using CPictFile

To learn the details of the PICT file format, see Tech-Note 27

You can use this class to read and write files whose type is `PICT`. These files contain "draw" type graphic images made with applications like MacDraw. A `PICT` file begins with 512 bytes of header information followed by a Macintosh picture.

Since CPictFile inherits its behavior from CFile, you need to use one of CFile's specification methods which file the `Open` method will open.

Variables

Variable	Type	Description
`header`	`Handle`	A handle to store the `PICT` header. Used internally.

Methods

IPictFile

```
procedure IPictFile;
void IPictFile (void);
```

Initialize the object. This method calls CFile's initialization method and allocates memory for the header.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the header, then dispose of the object.

ReadAll

```
function ReadAll: Handle;
Handle ReadAll (void);
```

Read the picture from the `PICT` file into contents. You can use the handle with Quickdraw routines that expect a `PicHandle` or you can pass it to CPicture's `SetMacPicture` method. If there is an error reading the file, this method returns NIL.

WriteAll

```
procedure WriteAll (contents: Handle);
void WriteAll (Handle contents);
```

Write a picture to a PICT file. The contents handle should be a `PicHandle`.

CPicture 49

Introduction

CPicture lets you display Macintosh pictures. The pictures are not editable.

Heritage

Superclass	CPanorama
Subclasses	None

Using CPicture

Use CPicture when you want to display a Macintosh picture. You can display the picture in a scroll frame or you can scale it to fit its frame. The picture is not editable. The picture is a standard Macintosh picture and is stored in an instance variable. You can look at the source code for this class to learn how to create your own panorama subclasses.

Variables

Variable	Type	Description
`macPicture`	`PicHandle`	Handle to the QuickDraw picture.
`scaled`	`Boolean`	TRUE if picture is scaled to fit in frame.
`isResPicture`	`Boolean`	TRUE if picture is read from a resource.
`ownsPicture`	`Boolean`	TRUE if this object should release memory for the picture when the object is destroyed.

Methods

Construction/Destruction

IPicture

```
procedure IPicture (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption);
void IPicture (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialize a picture. The arguments are identical to pane initialization.

Note

The descriptions of the other arguments are in CPane on page 347.

IViewRes

```
procedure IViewRes (rType: ResType; resID: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a picture from a resource template. `RType` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IPicture`. This method is inherited from CView.

To initialize a picture from a resource file, use a `'PctP'` resource.

IViewTemp

```
procedure IViewTemp (anEnclosure: CView;
   aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of this object. If the object owns the picture (`ownsPicture` is TRUE) the memory for the picture is released. If the picture comes from a resource (`isResPicture` is TRE), this method uses `HPurge` to purge the

`'PICT'` resource. If the picture is not from a resource, this method uses `KillPicture` to dispose of the picture.

Appearance methods

Draw

```
procedure Draw (var area: Rect);
void Draw (Rect *area);
```

Draw the picture. This method ignores the `area` parameter.

Accessing methods

SetMacPicture

```
procedure SetMacPicture (aMacPicture: PicHandle);
void SetMacPicture (PicHandle aMacPicture);
```

Use `aMacPicture` as the Macintosh picture for this object. If the picture comes from a resource, and the resource is purgeable, this method sets `ownsPicture` to FALSE. If the picture does not come from a resource, the picture is set to be unpurgeable, and `ownsPicture` is set to TRUE.

UsePICT

```
procedure UsePICT (PICTid: integer);
void UsePICT (short PICTid);
```

Use the `'PICT'` resource with ID `PICTid` as the Macintosh picture for this object. This method gets the resource and then callse `SetMacPicture`.

GetMacPicture

```
function GetMacPicture: PicHandle;
PicHandle GetMacPicture (void);
```

Return a handle to the Macintosh picture.

Calibration methods

SetScaled

```
procedure SetScaled (aScaled: Boolean);
void SetScaled (Boolean aScaled);
```

If `aScaled` is `TRUE`, the picture will be scaled to fit its frame.

GetScaled

```
function GetScaled: Boolean;
Boolean GetScaled (void);
```

Return `TRUE` if the picture is scaled.

ResizeFrame

```
procedure ResizeFrame (delta: Rect);
void ResizeFrame (Rect *delta);
```

Resize the picture's frame by the amount specified.

FrameToBounds

```
procedure FrameToBounds;
void FrameToBounds (void);
```

Make the frame of the picture the same size as the bounds.

CPNTGFile 50

Introduction

CPNTGFile is a class for reading and writing "paint" files like the kind MacPaint produces.

Heritage

Superclass	CDataFile
Subclasses	None

Using CPNTGFile

To learn the details of the PNTG file format, see Tech-Note 86

You can use this class to read and write files whose type is PNTG. These files contain "paint" or bitmapped graphic images made with applications like MacPaint. A PNTG file begins with 512 bytes of header information followed by a packed bitmap of the image.

Since CPNTGFile inherits its behavior from CFile, you need to use one of CFile's specification methods which file the Open method will open.

The reading and writing methods of this class use objects of class CBitMap. That class gives you an object-oriented way to work with Macintosh bitmaps.

Variables

Variable	Type	Description
header	Handle	A handle to store the PNTG header. Used internally.

Methods

IPNTGFile

```
procedure IPNTGFile;
void IPNTGFile (void);
```

Initialize the object. This method calls CFile's initialization method and allocates memory for the header.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the header, then dispose of the object.

ReadNewBitMap

```
function ReadNewBitMap (makePort: Boolean): CBitMap;
CBitMap *ReadNewBitMap (Boolean makePort);
```

Create a new object of class CBitMap and read the contents of this PNTG file into it. The `makePort` parameter is passed to CBitMap's `IBitMap` method. If it's true, `IBitMap` creates a Quickdraw grafport. If there was an error reading the image, this method returns `NIL`.

Note that this method creates a new bitmap every time you call it. The image is not stored in an instance variable.

WriteBitMap

```
procedure WriteBitMap (theBitMap: CBitMap);
void WriteBitMap (CBitMap *theBitMap);
```

Write an object of class CBitMap to a PNTG file.

CPrinter 51

Introduction

CPrinter is a class that handles standard Macintosh printing dialogs and calls the appropriate Print Manager routines.

Heritage

Superclass	CObject
Subclasses	None

Using CPrinter

The printer object manages communication between a document and the Macintosh print manager. Every document object can have a printer object associated with it. The document's `PrintPageOfDoc` method is the method that actually does the printing.

If your document is long, you may need to paginate your document (that is, divide it into pages). CPrinter lets you split your document into strips, each of which is a row or column of pages. At the intersection of a vertical strip and a horizontal strip, there is a page. Figure 51-1 shows how you might paginate graphic and text documents.

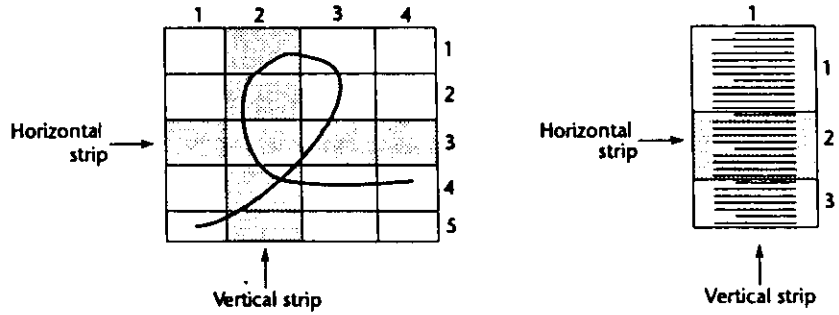

Figure 51-1 Paginating graphic and text documents

Note that strips don't have to be all the same size. Also, notice that each strip has a number. You need this number if you set the size or position of each strip individually.

CPrinter lets you set the size and position of strips in a variety of ways. Here are a few:

- To make all pages exactly the size the user chose in the **Page Setup...** dialog, use `SetStrips` on page 392.
- To insert a page break at a specific place in a document, use `SetHorizPageBreak` and `SetVertPageBreak` on page 393.
- To make all pages the same height or width, use `SetAllStripWidths` or `SetAllStripHeights` on page 393.
- To make a specific strip a specific height or width, use `SetStripWidth` or `SetStripHeight` on page 393.

If your document has more than one horizontal and vertical strip, you can choose the order the pages print in: horizontally (left to right first) or vertically (top to bottom first), as shown in Figure 51-2. Just send the printer the `SetPrintDir` message with either `printHoriz` or `printVert` as the argument.

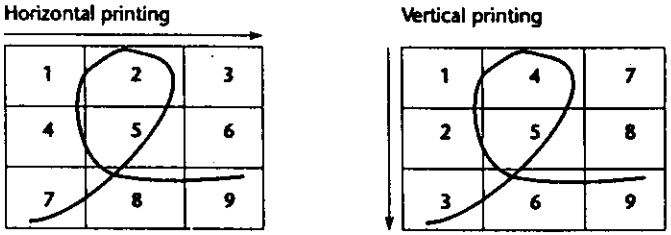

Figure 51-2 Printing horizontally and vertically

If you like, you can store a Macintosh print record with your document to preserve defaults. You can pass a handle to this record to the `IPrinter` method when you initialize your document. For methods in this class that return a Boolean value, TRUE means that the printer record stored with the printer object has changed. Ordinarily, you won't need to create a subclass of this class.

Variables

Variable	Type	Description
`itsDocument`	`CDocument`	Document using this Printer.
`macTPrint`	`THPrint`	Toolbox print record.
`printDirection`	`tPrintDirection`	The direction of printing when there is more than one strip.
`printMgrOpen`	`Boolean`	TRUE, if the Print Manager is open.
`printDocOpen`	`Boolean`	TRUE, if this printer is currently printing the document.
`printPageOpen`	`Boolean`	TRUE, if this printer is currently printing a page.
`savedResFile`	`integer`	The resource file that this application used before CPrinter opened the Print Manager.
`itsStripWidths`	`CRunArray`	List of strip widths.
`itsStripHeights`	`CRunArray`	List of strip heights.

Methods

Construction and destruction methods

IPrinter

```
function IPrinter (aDocument: CDocument;
   aMacTPrint: THPrint): Boolean;
```

```
Boolean IPrinter (CDocument *aDocument,
   THPrint aMacTPrint);
```

Initialize a printer object. `ADocument` is the document the printer object is associated with. `AMacTPrint` is a Macintosh print record handle. If `aMacTPrint` is `NIL`, this method creates a new print record. This method returns `TRUE` if it had to update the `aMacTPrint` record because it was incompatible. The document's initialization method initializes a printer object automatically if the printable parameter to `IDocument` is `TRUE`.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of a printer object.

Accessing methods

OpenPrintMgr

```
function OpenPrintMgr (fCheckFailure: Boolean):
   Boolean;
Boolean OpenPrintMgr (Boolean fCheckFailure);
```

Open the Print Manager to get ready to print. If `fCheckFailure` is TRUE and this method could not open the Print Manager, this method displays an alert telling the user to choose a printer with the Chooser and returns FALSE. This method returns TRUE if there was no error or if `fCheckFailure` is FALSE.

ClosePrintMgr

```
procedure ClosePrintMgr;
void ClosePrintMgr (void)
```

Close the Print Manager. This method aborts the printing of the current page and document, if this printer is in the middle of printing.

SetPrintDir

```
procedure SetPrintDir (aPrintDir: tPrintDirection);
void SetPrintDir (tPrintDirection aPrintDir);
```

Set the direction of printing, in case there is more than one horizontal and vertical strip. `TPrintDirection` can be `printHoriz` or `printVert`.

HavePagination

```
function HavePagination: Boolean;
Boolean HavePagination (void)
```

Return TRUE if this printer has any pagination information.

ResetPagination

```
procedure ResetPagination;
void ResetPagination (void)
```

Clear the current pagination, setting the number of horizontal and vertical strips to zero.

SetStrips

```
procedure SetStrips (numHStrips, numVStrips: integer);
void SetStrips (short numHStrips, short numVStrips);
```

Clear the current pagination and initialize it. This method sets the number of strips to `numHStrips` and `numVStrips` and sets the width (or height) of each strip to the width (or height) of the page size set in **Page Setup...**.

SetHorizPageBreak

```
procedure SetHorizPageBreak (vStripNum: integer,
   hPos: longint);
void SetHorizPageBreak (short vStripNum, long hPos);
```

Set a horizontal page break. `VStripNum` is the strip you're changing. `HPos` is the location of the page break in the frame coordinates of this printer's document.

SetVertPageBreak

```
procedure SetVertPageBreak (hStripNum: integer,
   vPos: longint);
void SetVertPageBreak (short hStripNum, long vPos);
```

Set a vertical page break. `HStripNum` is the strip you're changing. `VPos` is the location of the page break in the frame coordinates of this printer's document.

SetAllStripWidths

```
procedure SetAllStripWidths (aStripWidth: integer);
void SetAllStripWidths (short aStripWidth);
```

Sets the width all vertical page strips to `aStripWidth`.

SetAllStripHeights

```
procedure SetAllStripHeights (aStripHeight: integer);
void SetAllStripHeights (short aStripHeight);
```

Sets the height all horizontal page strips to `aStripHeight`.

SetStripWidth

```
procedure SetStripWidth (pageNum, aStripWidth:
   integer);
void SetStripWidth (short pageNum, short aStripWidth)
```

Sets the width of the vertical strip `pageNum` to `aStripWidth`.

SetStripHeight

```
procedure SetStripHeight (pageNum, aStripHeight:
   integer);
void SetStripHeight (short pageNum,
   short aStripHeight)
```

Sets the height of the horizontal strip `pageNum` to `aStripheight`.

GetStripCount

```
procedure GetStripCount (var hStrips, vStrips:
   integer);
void GetStripCount (short *hStrips, short *vStrips);
```

Get the number of horizontal and vertical page strips.

PageNumToStrips

```
procedure PageNumToStrips (pageNum: integer;
   var hStrip, vStrip: integer);
void PageNumToStrips (short pageNum, short *hStrip,
   short *vStrip)
```

Get the horizontal and vertical strips that page pageNum falls in.

GetPageStart

```
procedure GetPageStart (pageNum: integer;
   var startPos: LongPt);
void GetPageStart (short pageNum, LongPt *startPos)
```

Get the starting position of page pageNum in the frame coordinates of this printer's document.

GetPageArea

```
procedure GetPageArea (pageNum: integer;
   var pageArea: LongRect);
void GetPageArea (short pageNum, LongRect *pageArea)
```

Get the area of page pageNum.

GetPrintRecord

```
function GetPrintRecord: THPrint;
THPrint GetPrintRecord (void);
```

Return the Toolbox print record. This method calls PrValidate to update the record. You should treat the value that this method returns as read-only.

GetPageInfo

```
procedure GetPageInfo (var paperRect, pageRect:
   Rect; var hRes, vRes: integer);
void GetPageInfo (Rect *paperRect, Rect *pageRect,
   short *hRes, short *vRes);
```

Get information about the paper size and printable area of the page. The paperRect and pageRect are specified in dots. HRes and vRes specify the number of dots per inch.

DoPageSetup

```
function DoPageSetup: Boolean;
Boolean DoPageSetup (void);
```

Respond to a **Page Setup...** menu command. This method displays the standard job setup dialog, and returns TRUE if you made changes and pressed the OK button.

Printing methods

DoPrint

```
procedure DoPrint;
void DoPrint (void);
```

Respond to a **Print...** menu command. This method displays the standard print job dialog. If you click on the OK button, this method sends a PrintPageRange message.

PrintPageRange

```
procedure PrintPageRange (firstPage, lastPage:
   integer);
void PrintPageRange (short firstPage, short lastPage);
```

Print the specified range of the document associated with this printer object. The DoPrint method usually sends this message. Your application can bypass the print dialogs and send this message directly, but Tech Note 122 discourages this practice.

This method sends your document an AboutToPrint message to give it an opportunity to adjust the page range. Then, for each page in the page range, it sends your document a PrintPageofDoc message.

CRadioControl 52

Introduction

CRadioControl implements a standard Macintosh radio button.

Note

Earlier versions of The THINK Class Library used a different scheme to handle radio buttons. This class uses the dependent/provider mechanism implemented by CCollaborator to work with groups of radio buttons. The older class, CRadioButton is provided for backward compatibility, but not recommended.

Heritage

Superclass	CButton
Subclasses	None

Using CRadioControl

CRadioControl is a class that implements the standard Macintosh radio button. Since radio buttons always come in groups, a button must be a part of a radio group. The class CRadioGroupPane implements radio groups.

Like any other button, a radio button can have a command associated with it. Use the `SetClickCmd` method to set a radio button's command. You can also use any of the other CControl methods to manipulate the radio button.

See CRadioGroup pane for a discussion about working with groups of radio buttons.

Variables

This class has no instance variables.

Methods

IRadioControl

```
procedure IRadioControl (CNTLid: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void IRadioControl (short CNTLid, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialize a radio button from a CNTL resource. `CNTLid` is the resource ID for the radio button. `AnEnclosure` is the pane the radio button appears in. The enclosure should be a CRadioGroupPane. `ASupervisor` is the supervisor of the radio button. The supervisor should be a CRadioGroupPane.

INewRadioControl

```
procedure INewRadioControl (aWidth, aHeight: integer;
   aHEncl, aVEncl: integer; title: StringPtr;
   fVisible: Boolean, anEnclosured: CView,
   aSupervisor: CBureaucrat);
void INewRadioControl (short aWidth, short aHeight,
   short aHEncl, short aVEncl, StringPtr title,
   Boolean fVisible, CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialize a radio button from the parameters in the argument list. `AWidth` and `aHeight` are the width and height of the button in pixels. `AHEncl` and `aVEncl` are the horizontal and vertical position of the button within its enclosure. `Title` is the text to write beside the button. And if `fVisible` is TRUE, the window is drawn immediately after it's created

Note

The rest of the parameters are described under `IRadioControl` on page 398.

DoGoodClick

```
procedure DoGoodClick (whichPart: integer);
void DoGoodClick (short whichPart);
```

When the user presses and releases the mouse within the radio button, and the radio button was off, this method calls `SetValue` method to turn on the radio button. Note that `SetValue` in CControl sends a `BroadcastChange` message. The `ProviderChanged` method in CRadioGroupPane takes care of turning off the button that was on.

CRadioGroupPane 53

Introduction

CRadioGroupPane is a class that manages a group of radio buttons.

Note

Earlier versions of The THINK Class Library used a different scheme to handle radio buttons. This class uses the dependent/provider mechanism implemented by CCollaborator to work with groups of radio buttons. The older class, CRadioGroup is provided for backward compatibility, but not recommended.

Heritage

Superclass	CPane
Subclasses	None

Using CRadioGroupPane

A radio group pane is a pane specifically designed for grouping radio buttons of class CRadioControl. The radio group pane helps you make sure that only one radio button in a group is on. The button that is on in a radio group is called the **station**.

To add a button to a radio group pane, simply use the group as the button's supervisor when you initialize the button. The group pane assumes that all its subviews are radio buttons. When a radio button is selected, it sends a `BroadcastChange` message with `controlValueChanged` as the reason. The group pane turns off the previously selected button and sets `currentStation` to the newly selected button.

There are two ways you can respond to clicks in radio buttons. One way is to give each radio button a unique ID, and to use `SetStationID` to set the initial radio button. After that, let the radio group pane manage the radio

buttons. When you want to find out which button is the station, use `GetStationID`.

The other way is to give each radio button its own command. Use the `SetClickCmd` method that CRadioControl inherits from CButton to give a radio button its own command. When you click on a radio button to turn it on, CRadioControl's `DoGoodClick` method sends the button's supervisor a `DoCommand` message. If the radio button's supervisor is a plain CRadioGroupPane, the command should be handled by the group pane's supervisor. Or you can create a subclass of CRadioGroupPane with a `DoCommand` method to handle clicks in radio buttons.

Variables

Only CRadioGroupPane and its subclasses should access this instance variable. In THINK C, this variable is protected.

Variable	Type	Description
`currentStation`	`CRadioControl`	The currently selected radio button

Methods

Construction and destruction methods

IRadioGroupPane

```
procedure IRadioGroupPane (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption);
void IRadioGroupPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing);
```

Initialize a radio group pane. `ASupervisor` is the bureaucrat that owns the radio group pane. Typically, the supervisor is a pane or a window.

Note

The descriptions of the other arguments are in CPane on page 347.

Accessing methods

SetStationID

```
procedure SetStationID (aStationID: longint);
void SetStationID (long aStationID);
```

Change the current selection to the button with the specified ID number.

GetStationID

```
function GetStationID: longint;
long GetStationID (void);
```

Return the ID number of the currently selected radio button. Returns 0 if no station is selected.

Change notification method

ProviderChanged

```
procedure ProviderChanged (aProvider: CCollaborator;
   reason: longint; info: Ptr);
void ProviderChanged (CCollaborator *aProvider,
   long reason, void* info);
```

A radio button in this radio group pane has just been selected and sent a `BroadcastChange` message with `controlValueChanged` as the reason. This method turns off the previously selected button and sets `currentStation` to the newly selected button.

CResFile
54

Introduction

CResFile is an abstract class for working with Macintosh resource files.

Heritage

Superclass	CFile
Subclasses	None

Using CResFile

To learn about Macintosh resources and resource files, see Inside Macintosh I, Chapter 5, Inside Macintosh IV, Chapter 3, and TechNote 214.

If your application reads and writes resources make a subclass of CResFile and give it methods to access the resources. CResFile gives you methods to open and close resource files and to make the file the current resource file.

Since CResFile inherits its behavior from CFile, you need to use one of CFile's specification methods which file the Open method will open.

Variables

Variable	Type	Description
`refNum`	`Integer`	File system reference number of opened file

Methods

IResFile

```
procedure IResFile;
void IResFile (void);
```

Initialize the object. This method calls CFile's initialization method and initializes `refNum` to 0.

Open

```
procedure Open (permission: SignedByte);
void Open (SignedByte);
```

Open the resource file with the specified permission. If the file can't be opened, this method calls `FailResError`. Remeber that you need use one of CFile's specification methods to specify which file to open.

Close

```
procedure Close;
void Close (void);
```

Close this resource file. This method calls `FailOSErr` if there was a problem closing the file.

MakeCurrent

```
procedure MakeCurrent;
void MakeCurrent (void);
```

This method makes the resource file the current resource file.

IsOpen

```
function IsOpen: Boolean;
Boolean IsOpen (void);
```

Returns TRUE if the resource fork of this file is open.

Update

```
procedure Update;
void Update (void);
```

Updates this resource file by writing out the changed resource.

HasResFork

```
function HasResFork: Boolean;
Boolean HasResFork (void);
```

Returns TRUE if this file has a resource fork. The file must have been previously specified.

CreateNew

```
procedure CreateNew (creator, fType: OSType);
void CreateNew (OSType creator, OSType fType)
```

Creates a new resource file. If the file already exists but has no resource fork, this method gives it a resource fork.

CRunArray 55

Introduction

CRunArray implements a dynamic array of long integers that can conserve space.

Heritage

Superclass	CArray
Subclasses	None

Using CRunArray

CRunArray implements a dynamic array of long integers. It conserves space if your array contains lots of sequences of entries with the same value. These sequences are called runs. A run array consists of runs, which contain a value and the number of consecutive entries that have that value. For example, Figure 8-1 shows how a traditional array and a run array would store the same values.

Traditional Array

Index	Value
1	2
2	64
3	64
4	64
5	45

Run Array

Run	#Entries	Value
1	1	2
2	3	64
3	1	45

Figure 8-1 A traditional array and a run array

To put an entry into the array, you can choose between `SetValue` and `InsertValue`. `SetValue` replaces an entry with a new entry. `InsertValue` inserts a run of values into the array. To delete an entry, use

DeleteValue. To sum a range of entries in the array, use the methods SumRange and FindSum.

When used with a run array, many CArray methods operate on runs of entries, not on individual entries. This table shows you some of those CArray methods and gives the CRunArray method you should use instead.

CArray method	CRunArray method
GetItem	GetValue
SetItem	SetValue
InsertAtIndex	InsertValue
DeleteItem	DeleteValue

In a CRunArray, the instance variable numItems contains the number of runs in the array. To find the number of entries, use the method GetNumItems, instead.

Variables

Variable	Type	Description
itemCount	longint	Number of entries in the array.
hRuns	tRunHndl	Handle to runs. Same as hItems.

Methods

Creation method

IRunArray

```
procedure IRunArray;
void IRunArray;
```

Initialize the array. The number of items and runs is set to 0 (zero).

Accessing method

GetNumItems

```
function GetNumItems: longint;
long GetNumItems (void);
```

Return the number of items in the array.

Insertion and deletion methods

InsertValue

```
procedure InsertValue (item, value, count: longint);
void InsertValue (long item, long value, long count);
```

Insert a run of values into the array. Item is the index to start the run at. If you specify an index beyond the end of the array, the run is added to the

end of the array. Value is the value for all in the entries in this run. Count is the length of the run.

SetValue

```
procedure SetValue (index, value: longint);
void SetValue (long index, long value);
```

Set the value of the entry at index to value.

DeleteValue

```
procedure DeleteValue (index: longint);
void DeleteValue (long index);
```

Delete the entry at index. All the entries following index move up one position.

DeleteAll

```
procedure DeleteAll;
void DeleteAll (void);
```

Delete all the entries in this array.

Membership method

GetValue

```
function GetValue (index: longint): longint;
long GetValue (long index);
```

Return the value of the entry at index.

Summing methods

SumRange

```
function SumRange (startIndex, endIndex: longint):
   longint;
long SumRange (long startIndex, long endIndex);
```

Return the sum of the items between startIndex and endIndex, inclusive.

FindSum

```
function FindSum (aSum: longint): longint;
long FindSum (long aSum);
```

Return the index of the first entry such that all the entries from 1 to that entry have a sum equal to or greater than sum.

Run-handling methods

CArray uses these methods internally to manipulate runs. You should need to use them only if you are creating a subclass of CRunArray. In THINK C, they are protected.

FindRun

```
procedure FindRun (itemIndex: longint;
   var runIndex, firstInRun: longint);
void FindRun (long itemIndex, long *runIndex,
   long *firstInRun);
```

Return the number of the run that contains the entry at `itemIndex`. This method sets `runIndex` to the run number and `firstInRun` to the index of the first entry in the run. If `itemIndex` is not in the array, this method sets `runIndex` and `firstInRun` to `BAD_INDEX`.

InsertRun

```
procedure InsertRun (index, runLength, value:
   longint);
void InsertRun (long index, long runLength,
   long value);
```

Insert a new run into the array. `Index` is the index of the first entry in the run. `RunLength` is the number of entries in the run. `Value` is the value of all the entries in the run.

DeleteRun

```
procedure DeleteRun (runIndex: longint);
void DeleteRun (long runIndex);
```

Delete the run at number `runIndex` from the array.

CScrollBar 56

CScrollBar implements a standard Macintosh scroll bar.

Heritage

Superclass	CControl
Subclasses	None

Using CScrollBar

This class implements a Macintosh scroll bar. To make scroll bars easier to use, this class distinguishes between a mouse click in an indicator (the scroll box or the "thumb") and a click in any other part of the scroll bar. Both behaviors are implemented in the `DoClick` method of the CControl class.

When you click in any part other than an indicator, the `DoClick` method calls the Toolbox routine `TrackControl` with an action procedure, or **action proc**. The action procedure is the routine that adjusts what the scroll bar controls. In most cases, the scroll bar controls a panorama. To set the action proc, use the `SetActionProc` method inherited from CControl.

When you click in the indicator, the `DoClick` method sends a `DoThumbDragged` message to the scroll bar. This method calls a **thumb function** that you provide. The thumb function is the routine that adjusts whatever the scroll bar controls. Thumb functions are unique to scroll bars. To set the thumb function, use the `SetThumbFunc` method.

Usually, you'll use a scroll bar to control a pane. The class CScrollPane is a scrollable pane (a panorama) with one or two scroll bars. The CScrollPane class handles the usual cases, so you don't have to provide an action proc or a thumb function.

Variables

Variable	Type	Description
theOrientation	Orientation	Which way the scroll bar lies, horizontal or vertical.
theThumbFunc	ProcPtr	Function to call after a thumb drag.

Methods

Construction and destruction methods

IScrollBar

```
procedure IScrollBar (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   anOrientation: Orientation;
   aLength, aHEncl, aVEncl: integer);
void IScrollBar (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   Orientation anOrientation,
   short aLength, short aHEncl, short aVEncl);
```

Initialize a scroll bar. AnEnclosure is the pane or window the scroll bar belongs to. ASupervisor is the scroll bar's supervisor in the chain of command. Orientation is either HORIZONTAL or VERTICAL. ALength is the length of the scroll bar. AHEncl and aVEncl are the horizontal and vertical position of the upper left corner of the scroll bar.

Accessing methods

SetThumbFunc

```
procedure SetThumbFunc (aThumbFunc: ProcPtr);
void SetThumbFunc (VoidFunc aThumbFunc);
```

Set aThumbFunc to be the scroll bar's thumb function. The default DoClick method for controls sends a DoThumbDragged message to the control when the user moves an indicator in a control. The DoThumbDragged method for scroll bars calls the thumb function. You should declare a Pascal thumb function like this:

```
procedure MyThumbFunc (theControl: CControl;
   delta: integer);
```

And you should declare a C thumb function like this:

```
void MyThumbFunc (CControl *theControl,
   short delta);
```


TheControl is the control whose indicator was moved. Delta is the amount by which the value changed. To get the current value of the control, you can send it a GetValue message.

Drawing methods

Draw

```
procedure Draw (var area: Rect);
void Draw (Rect *area);
```

Draw the scroll bar. If the scroll bar is active, this method draws it the normal way. If the scroll bar is inactive, this method draws only the frame of the scroll bar.

Activate

```
procedure Activate;
void Activate (void);
```

Activate the scroll bar.

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

Deactivate the scroll bar.

Click response methods

DoClick

```
procedure DoClick (hitPt: Point;
   modifierKeys: integer; when: longint);
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

Handle a click in the scroll bar. If the scroll bar's enclosure is a scroll pane, send an AdjustScrollMax message to the scroll pane.

DoThumbDragged

```
procedure DoThumbDragged (delta: integer);
void DoThumbDragged (short delta);
```

If the scroll bar has a thumb function associated with it, call it with the scroll bar and delta as arguments. The default DoClick method for controls sends a DoThumbDragged message to the control when the user moves an indicator in a control. See SetThumbFunc on page 410.

CScrollPane 57

Introduction

CScrollPane implements a pane with scroll bars that control a panorama.

Heritage

Superclass	CPane
Subclasses	None

Using CScrollPane

A scroll pane is a pane with a panorama and scroll bars to control what is being displayed in the pane. Most of your applications will use a scroll pane that occupies most of the window. After creating a scroll pane, you can send it a `FitToEnclFrame` message to make it as big as the window.

All you have to do to use a scroll pane is install a panorama with the `InstallPanorama` method. The scroll pane uses the scale of the panorama for the values of the scroll bars.

The scroll bars and panorama do not communicate directly. Mouse clicks in the scroll bars are reported to the scroll pane, which then tells the panorama how to scroll or shift its image. Similarly, changes in the panorama which would affect the scroll bars are reported to the scroll pane, which then adjusts the scroll bars.

Variables

Variable	Type	Description
itsPanorama	CPanorama	The scrollable view. The "content" of the scroll pane.
itsHorizSBar	CScrollBar	The scroll pane's horizontal scroll bar.
itsVertSBar	CScrollBar	The scroll pane's vertical scroll bar.
itsSizeBox	CSizeBox	The scroll pane's size box.
hExtent	longint	For internal use.
vExtent	longint	For internal use.
hUnit	integer	For internal use.
vUnit	integer	For internal use.
hSpan	integer	For internal use.
vSpan	integer	For internal use.
hStep	integer	Number of horizontal units to scroll by when the user clicks on an arrow.
vStep	integer	Number of vertical units to scroll by when the user clicks on an arrow.
hOverlap	integer	Number of units to overlap when the user clicks in a page region.
vOverlap	integer	Number of units to overlap when the user clicks in a page region.

Methods

Construction and destruction methods

IScrollPane

```
procedure IScrollPane (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight, aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption;
   hasHoriz, hasVert, hasSizeBox: Boolean)
void IScrollPane (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   Boolean hasHoriz, Boolean hasVert,
   Boolean hasSizeBox);
```

Initialize a scroll pane. All but the last three arguments are identical to the arguments to `IPane`. If `hasHoriz` is TRUE, the scroll pane has an horizontal scroll bar. If `hasVert` is TRUE, the scroll pane has a vertical scroll bar. If `hasSizeBox` is TRUE, the scroll pane draws a size box in the lower-right corner of the pane.

Note

The descriptions of the other arguments are in CPane on page 347.

IViewRes

```
procedure IViewRes (rType: ResType; resID: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a scroll pane from a resource template. `RTtype` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` and `aSupervisor` are the same as for `IScrollPane`. This method is inherited from CView.

To initialize a scroll pane from a resource file, use a `'ScPn'` resource.

IViewTemp

```
procedure IViewTemp (anEnclosure: CView;
   aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

This method is used internally for initializing from a resource template. Each subclass of CView overrides this method to use its own resource template.

Accessing methods

InstallPanorama

```
procedure InstallPanorama (aPanorama: CPanorama);
void InstallPanorama (CPanorama *aPanorama);
```

Establish `aPanorama` as the panorama associated with this scroll pane. This method sends `AdjustScrollMax` and `Calibrate` messages to the scroll pane.

SetSteps

```
procedure SetSteps (aHStep, aVStep: integer);
void SetSteps (short aHStep, short aVStep);
```

Set the amount to scroll when the user clicks on the arrows of a scroll bar. The units are in the panorama's units.

GetSteps

```
procedure GetSteps (var theHStep, theVStep: integer);
void GetSteps (short *theHStep, short *theVStep);
```

Get the amount to scroll when the user clicks on the arrows of a scroll bar. The units are in the panorama's units.

SetOverlaps

```
procedure SetOverlaps (aHOverlap, aVOverlap: integer);
void SetOverlaps (short aHOverlap, short aVOverlap);
```

Set the amount of overlap when the user clicks in the page (gray) regions of the scroll bar. The units are in the panorama's units.

GetInterior

```
procedure GetInterior (var theInterior: LongRect);
void GetInterior (LongRect *theInterior);
```

Get the interior of the scroll pane. The interior excludes the space that the scroll bars occupy.

Scroll bar maintenance methods

AdjustScrollMax

```
procedure AdjustScrollMax;
void AdjustScrollMax (void);
```

Adjust the maximum value of the scroll bars from the extent and frame size of the panorama.

Calibrate

```
procedure Calibrate;
void Calibrate (void);
```

Adjust the scroll bar's thumb when the position of the frame of the panorama changes.

ChangeSize

```
procedure ChangeSize (delta: Rect; redraw: Boolean);
void ChangeSize (Rect *delta, Boolean redraw);
```

Change the size of a scroll pane. Each component of the `delta` rectangle specifies how each side will change. Positive values mean down and to the right. Negative values mean up and to the left. If `redraw` is `true`, the scroll pane is redrawn on the next update event.

Scroll performance methods

DoHorizScroll

```
procedure DoHorizScroll (whichPart: integer);
void DoHorizScroll (short whichPart);
```

Scroll horizontally. `WhichPart` specifies which part of the scroll bar was hit. This method sends a `DoScroll` message.

DoVertScroll

```
procedure DoVertScroll (whichPart: integer);
void DoVertScroll (short whichPart);
```

Scroll vertically. `WhichPart` specifies which part of the scroll bar was hit. This method sends a `DoScroll` message.

DoThumbDrag

```
procedure DoThumbDrag (hDelta, vDelta: integer);
void DoThumbDrag (short hDelta, short vDelta);
```

Adjust the panorama when the scroll box (thumb) has been dragged. This method sends a `DoScroll` message to the panorama.

DoScroll

```
procedure DoScroll (hDelta, vDelta: longint);
void DoScroll (long hDelta, long vDelta);
```

Scroll the panorama belonging to this pane by `hDelta` units horizontally and `vDelta` units vertically. The units are given in the panorama's coordinates. All the other methods in this class call this method to handle scrolling.

Functions

Note that these are procedures, not methods.

SBarActionProc

```
procedure SBarActionProc (macControl: ControlHandle;
   whichPart: integer);
pascal void SBarActionProc (ControlHandle
   macControl, short whichPart);
```

The Toolbox `TrackControl` routine calls this function continuously while the mouse is down in any part of the scroll bar except the scroll box (thumb). This function sends the scroll bar's supervisor (the scroll pane) `DoHorizScroll` and `DoVertScroll` messages that actually scroll the panorama.

SBarThumbFunc

```
procedure SBarThumbFunc (theSBar: CScrollBar;
   delta: integer);
void SBarThumbFunc (CScrollBar *theSBar, short delta);
```

The scroll bar's `DoThumbDragged` method calls this routine after the scroll box of a scroll bar has been moved. A control's `DoClick` method sends a `DoThumbDragged` message when the user moves the indicator of a control. This function sends the scroll bar's supervisor (the scroll pane) a `DoThumbDrag` message.

CSelector 58

Introduction

CSelector is an abstract class for drawing panes with several items that users can choose from. A tool palette is an example of a selector.

Heritage

Superclass	CPanorama
Subclasses	CGridSelector

Using CSelector

CSelector is an abstract class that defines the basic behavior of a pane that lets you choose from several items. A good example of a selector is a tool palette or a pattern palette. The THINK Class Library includes CGridSelector, which is a class for implementing those kinds of palettes, and its descendants CPatternGrid, for displaying the standard patterns, and CCharGrid, for displaying characters in a grid. You can use these classes directly for pattern palettes and tool palettes. If you want to implement a different kind of selector, you'll need to create a subclass of CSelector.

A selector works like menu without command numbers. Each selector has a **command base**, which is like a menu ID. You set the command base when you create a selector, or you can set it once it's been created. A selector contains **items**. You specify the number of items when you create the selector.

Every selector needs to have a `Draw` method to display the items. How the items appear in the selector and how they're arranged is up to you. You will also need to override the `FindItem` method to determine which item you clicked on, and you'll need to override the `HiliteItem` method highlight the selected item. If you want your selector to respond to double-click's you'll need to override the `DoDoubleClick` method as well.

Variables

Variable	Type	Description
numItems	integer	The number of items to choose from.
selection	integer	The currently selected item.
commandBase	integer	The base value for converting selections into command numbers.

Methods

Construction and destruction methods

ISelector

```
procedure ISelector (anEnclosure: CView;
   aSupervisor: CBureaucrat;
   aWidth, aHeight: integer;
   aHEncl, aVEncl: integer;
   aHSizing, aVSizing: SizingOption;
   aNumItems, aSelection, aCommandBase: integer);
void ISelector (CView *anEnclosure,
   CBureaucrat *aSupervisor,
   short aWidth, short aHeight,
   short aHEncl, short aVEncl,
   SizingOption aHSizing, SizingOption aVSizing,
   short aNumItems, short aSelection,
   short aCommandBase);
```

Initialize a selector. The first eight arguments to this routine are identical to the ones for IPane.

ANumItems specifies the number of items in this selector. ASelection is the initial item. ACommandBase is the base value for converting the selected item into a command number.

Note

The descriptions of the other arguments are in CPane on page 347.

Mouse methods

DoClick

```
procedure DoClick (hitPt: Point;
   modifierKeys: integer; when: longint);
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

This method converts a click into a command number and sends it to the selector's supervisor in a DoCommand message. DoClick sends the selector a FindItem message to find out which item got the click.

Once it has an items, DoClick uses the same scheme as CBartender to build a command number.. It puts the command base in the high word of a long integer, the item number in the low word, and negates the resulting long integer.

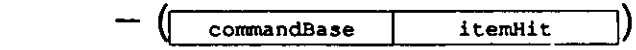

Figure 58-1 How DoClick builds the command number.

If the new selection is not the same as the current selection, this method sends a ChangeHilite message to the selector. If the new selection is the same as the current selection, and there was a double-click, this method sendsa DoDoubleClick message to the selector.

Since most of the work is done in methods the you must override, your CSelector subclass shouldn't need to override this method. You might want to override this method to handle things like triple-clicks.

HitSamePart

```
function HitSamePart (pointA, pointB: Point): Boolean;
Boolean HitSamePart (Point pointA, Point pointB);
```

Returns TRUE if the mouse went down in the same item, FALSE otherwise. The default method checks to see if FindItem(pointA) and FindItem(pointB) return the same item. Your CSelector subclass should not need to override this method.

Accessing methods

ChangeSelection

```
procedure ChangeSelection (aSelection: integer);
void ChangeSelection (short aSelection);
```

Changes the selection from the current selection to aSelection. If aSelection is not the same as the current selection, this method turns off the highlighting of the current selection and turns on the highlighting of the new selection. Your subclass should not need to override this method.

GetSelection

```
function GetSelection: integer;
short GetSelection (void);
```

Return the current selection. You shouldn't need to override this method.

SetCommandBase

```
procedure SetCommandBase (aCommandBase: integer);
void SetCommandBase (short aCommandBase);
```

This method sets the selector's command base. CSelector uses the command base to build a command number from the item hit. Your subclass should not need to override this method.

GetCommandBase

```
function GetCommandBase: integer;
short GetCommandBase (void);
```

Return the value of the selector's command base. Your subclass should not need to override this method.

HiliteItem

```
procedure HiliteItem (theItem: integer;
   state: HiliteState);
void HiliteItem (short theItem, HiliteState state);
```

Hilight the specified item. `HiliteItem` can take on of three values for the `state` parameter:

Hilite state value	Behavior
`hiliteOFF`	Turn highlighting off for the item.
`hiliteON`	Turn highlighting on for the item.
`hiliteDYNAMIC`	Flash the selected item without selecting or deselecting.

It's up to you how you highlight the items of your selector. In most cases, it's sufficient to invert the rectangle that encloses the specified item. When highlighting is on, the item should be inverted. When highlighting is off, the item should appear normally. Dynamic highlighting is used when the selector is used as a menu. See the CSelectorMDEF class.

FindItem

```
function FindItem (hitPt: Point): integer;
short FindItem (Point hitPt);
```

Determine which item corresponds to a mouse down at a specified point. It's up to you how your selector arranges and displays its items. Typically, items are arranged in a grid or a table. Your subclass must override this method.

DoDoubleClick

```
procedure DoDoubleClick;
void DoDoubleClick (void);
```

Respond to a double-click. The first click will have set the selection, so the double-click will pertain to that item. If your selector subclass responds to double-clicks, you should override this method.

CSelectorMDEF 59

Introduction

CSelectorMDEF is a class that lets you use descendants of CSelector as custom menus.

Heritage

Superclass	CPaneMDEF
Subclasses	None

Using CSelectorMDEF

CSelectorMDEF is a class that handles menu selection for custom menus based on CSelector panes. If you pass CSelectorMDEF's initialization method a descendant of CSelector, you'll be able to use it as a custom menu. CSelectorMDEF's ChooseItem method takes care of selecting items from the CSelector. If you pass `ISelectorMDEF` a descendant of CTearOffMenu, you can use your custom menu as a tear-off menu.

Variables

This class has no instance variables.

Methods

Construction and destruction methods

ISelectorMDEF

```
procedure ISelectorMDEF (MDEFid: integer;
   aPane: CPane; aTearOffMenu: CTearOffMenu);

void ISelectorMDEF (short MDEFid, CPane *aPane,
   CTearOffMenu *aTearOffMenu);
```

Initialize the selector MDEF. This method passes the arguments to CPaneMDEF's initialization method. `APane` must be a selector pane descended from CSelector. CTearOffMenu should be a descendant of CTearOffMenu where the custom menu will be displayed when it's torn off.

ChooseItem

```
procedure ChooseItem (macMenu: MenuHandle;
   menuRect: Rect; hitPt: Point;
   var whichItem: integer);
void ChooseItem (MenuHandle macMenu, Rect *menuRect,
   Point hitPt, short *whichItem);
```

This method handles menu selection for custom menus.

If the `hitPt` is within the `menuRect`, this method sets up the QuickDraw drawing environment to match the pane's drawing environment. The QuickDraw origin is set so the point (0, 0) is the top left of the pane. `ChooseItem` then sends a `FindItem` message to the pane associated with this object.

CSizeBox 60

Introduction

CSizeBox implements a Macintosh grow icon.

Heritage

Superclass	CPane
Subclasses	None

Using CSizeBox

In most cases, you will not need to use this class yourself. It is used by CScrollPane to draw a Macintosh grow icon at the lower left corner of any pane, not just in the lower right corner of a window.

The first version of the THINK Class Library used an 'SICN' resource to draw the grow icon. If the `useSICN` flag is TRUE, the `Draw` method uses the resource, otherwise it calls the Toolbox routine `DrawGrowIcon`.

Variables

Variable	Type	Description
`useSICN`	`Boolean`	TRUE, if this size box uses a `SICN` resource instead of calling `DrawGrowIcon`.

Methods

Construction and destruction methods

ISizeBox

```
procedure ISizeBox (anEnclosure: CView;
   aSupervisor: CBureaucrat);
void ISizeBox (CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialize a size box. This method places a pane containing a size box at the lower right of its enclosure. The sizing options for a size box are `sizFIXEDRIGHT` and `sizFIXEDBOTTOM`. By default, `useSICN` is FALSE.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of a size box.

Appearance methods

Draw

```
procedure Draw (var area: Rect);
void Draw (Rect *area);
```

Draw a size box. If the pane is active, this method draws the size box. If the pane is inactive, this method draws a white rectangle. If `useSICN` is FALSE, this method uses the Toolbox's grow icon, otherwise it uses a `'SICN'` resource to draw the grow icon.

Activate

```
procedure Activate;
void Activate (void);
```

The enclosure the size box belongs to is becoming active. The default method sends a `Draw` message to the size box

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

The enclosure the size box belongs to is becoming inactive. The default method sends a `Draw` message to the size box.

Class resources

If `useSICN` is TRUE, this class uses an `'SICN'` resource to draw the grow icon.

Resource	Description
`SICN 200`	The grow icon

CStack 61

Introduction

CStack implements a stack of objects.

Heritage

Superclass	CCluster
Subclasses	None

Using CList

Use an object of class CStack when you need to maintain a stack of objects. You can use the iteration methods that CStack inherits from CCluster to apply functions to each item in the stack,

Variables

This class has no instance variables.

Methods

IStack

```
procedure IStack;
void IStack (void);
```

Initialize the stack. This method calls the CCluster's initialization method.

Push

```
procedure Push (theObect: CObject);
void Push (CObject *theObject);
```

Push theObject on the stack.

Pop

```
function Pop : CObject;
CObject *Pop;
```

Pop the item from the top of the stack. If the stack is empty, this method returns `NIL`.

CSwitchboard 62

Introduction

CSwitchboard is the class that processes Macintosh Toolbox events and sends the appropriate messages to objects in the THINK Class Library. There is normally only one instance of this class.

Heritage

Superclass	CObject
Subclasses	None

Using CSwitchboard

The single instance of this class handles all the Macintosh Toolbox events and sends messages to objects. The application's Run method repeatedly sends `ProcessEvent` messages to this object to dispatch messages to the objects that make up your application.

The application initialization method `IApplication` creates the single instance of CSwitchboard. The switchboard is stored in the application's `itsSwitchboard` instance variable.

Note

You need to subclass CSwitchboard only if your application handles `appl Evt`, `app2Evt`, or `app3Evt` events. Then you should override the `DoOtherEvent` method.

Variables

`mouseRgn`	`RgnHandle`	Argument for `WaitNextEvent` to handle cursor adjustment.

Methods

Initialization methods

ISwitchboard

```
procedure ISwitchboard;
void ISwitchboard (void);
```

Initialize the switchboard. The application's `IApplication` method creates the switchboard and sends it this message. This method also installs one handler for all AppleEvents. The handler sends the switchboard a `DoAppleEvent` message.

Mouse methods

DoMouseDown

```
procedure DoMouseDown (macEvent: EventRecord);
void DoMouseDown (EventRecord *macEvent);
```

The user pressed the mouse button. This method sends a `DispatchClick` message to the desktop, and stores the event in the global `gLastMouseDown`.

DoMouseUp

```
procedure DoMouseUp (macEvent: EventRecord);
void DoMouseUp (EventRecord *macEvent);
```

The user released the mouse button. This message sends a `DoMouseUp` message to the last view hit. Since mouse ups always follow a mouse down, it's not important where the mouse came up, but the message must be sent to the same object that handled the mouse down. This method stores the event in the global `gLastMouseUp`.

Key methods

DoKeyEvent

```
procedure DoKeyEvent (macEvent: EventRecord);
void DoKeyEvent (EventRecord *macEvent);
```

The user pressed or released a key. If the user holds down the Command key and presses a key at the same time, this method uses the Toolbox routine `MenuKey` to find the menu equivalent. If there is a menu equivalent, this method sends a `DoCommand` message to the gopher. If there is no menu equivalent, this method sends a `DoKeyDown` message to the gopher. For other key events, this method send a `DoKeyDown`, `DoKeyUp`, or `DoAutoKey` messages to the gopher.

Disk methods

DoDiskEvent

```
procedure DoDiskEvent (macEvent: EventRecord);
void DoDiskEvent (EventRecord *macEvent);
```

This method calls the Toolbox routine `DIBadMount` to mount a disk. This is the only event that the switchboard handles directly.

Window event methods

DoUpdate

```
procedure DoUpdate (macEvent: EventRecord);
void DoUpdate (EventRecord *macEvent);
```

This method sends an `Update` message to the window specified in the event record.

DoActivate

```
procedure DoActivate (macEvent: EventRecord);
void DoActivate (EventRecord *macEvent);
```

This method sends an `Activate` message to the window specified in the event record.

DoDeactivate

```
procedure DoDeactivate (macEvent: EventRecord);
void DoDeactivate (EventRecord *macEvent);
```

This method sends an `Deactivate` message to the window specified in the event record.

Suspend/Resume methods

DoSuspend

```
procedure DoSuspend (macEvent: EventRecord);
void DoSuspend (EventRecord *macEvent);
```

The application is about to be switched to the background under MultiFinder. This method sends a `Suspend` message to your application.

DoResume

```
procedure DoResume (macEvent: EventRecord);
void DoResume (EventRecord *macEvent);
```

The application is about to be switched to the foreground under MultiFinder. This method sends a `Resume` message to your application.

Event processing methods

DoOtherEvent

```
procedure DoOtherEvent (macEvent: EventRecord);
void DoOtherEvent (EventRecord *macEvent);
```

If your application handles `app1Evt`, `app2Evt`, or `app3Evt` events, override this method.

DoIdle

```
procedure DoIdle (macEvent: EventRecord);
void DoIdle (EventRecord *macEvent);
```

This method is invoked during null events. This method sends an `Idle` message to your application. The application uses its `Idle` message to perform periodic tasks.

DoHighLevelEvent

```
procedure DoHighLevelEvent (macEvent: EventRecord);
void DoHighLevelEvent (const EventRecord* macEvent);
```

Handle a high level event. This method assumes all high level events are AppleEvents. If you use a high level event that isn't an AppleEvent, you must override this method. Your method should call `inherited DoHighLevelEvent` to handle AppleEvents.

DoAppleEvent

```
function DoAppleEvent (macEvent, theReply: AppleEvent;
   refCon: longint): OSErr;
OSErr DoAppleEvent (AppleEvent *macEvent,
   AppleEvent *theReply, long refCon);
```

Respond to an AppleEvent. This method packages the AppleEvent and the default reply into a CAppleEvent object and sends it to the gopher. If an exception occurs, this method returns the error causing the exception. The exception does not propagate beyond this method.

This method calls sends the application a `PackageAppleEvent` message to package the event and its default reply into a CAppleEvent object. If you subclass CAppleEvent, override this method.

AppleEventIdle

```
function AppleEventIdle (macEvent: EventRecord;
   var sleepTime: longint; mouseRgn: RgnHandle):
   Boolean;
Boolean AppleEventIdle (EventRecord *macEvent,
   long *sleepTime, RgnHandle *mouseRgn);
```

The switchboard's default AppleEvent idle procedure sends the switchboard this message. The Toolbox functions `AEInteractWithUser` and `AESend` use the idle procedure to respond to an event while waiting for the user to respond to an AppleEvent. The possible events are null, update, OS, or activate events.

To use a different idle procedure, set the `idleProc` instance variable in the CAppleEvent object, described on page 157.

This method returns TRUE if the user aborted by pressing Command-. (Command-Period), and FALSE if the user wants to continue waiting.

ProcessEvent

```
procedure ProcessEvent;
void ProcessEvent (void);
```

This method is the heart of your application's event loop. This method gets an event and sends a message to the switchboard to handle it. Before processing the event, `ProcessEvent` sends a `DispatchCursor` message to the application to adjust the cursor.

GetAnEvent

```
function GetAnEvent (macEvent: EventRecord): Boolean;
Boolean GetAnEvent (EventRecord *macEvent);
```

Get the next event in the event queue. This method calls one of the Toolbox routines `GetNextEvent` or `WaitNextEvent` to get an event and returns the result. If you need to do something to an event before the switchboard handles it, override this method. Your method should call `inherited GetAnEvent` and then do what you want with the event.

DispatchEvent

```
procedure DispatchEvent (macEvent: EventRecord);
void DispatchEvent (EventRecord *macEvent)
```

This method is the main event dispatcher. Depending on the event, it sends an appropriate message back to itself to handle the event.

CTask 63

Introduction

CTask is an abstract class for implementing undoable actions.

Heritage

Superclass	CObject
Subclasses	CMouseTask
	CTextEditTask
	CTextStyleTask

Using CTask

A task is an abstract class for implementing undoable actions. If you want your application to be able to undo an action, you need to define a task subclass for each action.

You can use a task two ways. You can perform your action, create a task object, store enough information in it for its Undo method to undo the action, and send it in a `Notify` message to your supervisor (usually the document).

The second way is similar to the first, but you also implement a `Do` method that performs the action. So you create a task, send it a `Do` message to perform the action, and then send it in a `Notify` message to your supervisor. Your `Do` method stores enough information in the task's instance variables to undo the action.

When you notify a document that you've performed a task, it stores the task in the instance variable `lastTask`. When you choose **Undo** from the **Edit** menu, the document's `DoCommand` method sends an Undo message to that task.

Every task subclass has a string in the `STR# 130` resource used for the wording of the **Undo/Redo** command. Tasks have an instance variable that is the index of its string in the `STR#` resource. The document's

UpdateUndo method takes care of the wording of the **Undo/Redo** command.

Here's an example. Suppose you've defined a subclass of CTask to change the font in an edit text pane. Before passing the command on to the edit pane's DoCommand method, you create a task and store the current font in an instance variable. After you pass the font command to the edit text pane, you send the task in a Notify message to the document.

Your Undo method would simply send the font change command to the document. Since the command goes through the regular command chain, your DoCommand method would create a task to let you undo what you were undoing.

Variables

Variable	Type	Description
nameIndex	integer	Index of the **Undo/Redo** string in the STR# 130 resource.
undone	Boolean	Is this task undone?

Methods

Initialization methods

ITask

```
procedure ITask (aNameIndex: integer);
void ITask (short aNameIndex);
```

Initialize a task object. ANameIndex is the index of the task's **Undo** string in the STR# 130 resource. Your subclass's initialization method should call this method in addition to any other initialization it does. If your task subclass allocates memory, you'll also need to implement a Free method to release that memory.

Accessing methods

GetNameIndex

```
function GetNameIndex: integer;
short GetNameIndex (void);
```

Get the task's index. This method is used by the document's UpdateUndo method.

IsUndone

```
function IsUndone: Boolean;
Boolean IsUndone (void);
```

Return whether this task is undone.

Action methods

DoTask/Do

```
procedure DoTask;
void Do (void);
```

Perform a task. The default method does nothing. If you want to use a task to implement an action which is not necessarily undoable, your subclass should override this method. The **Undo/Redo** mechanism doesn't send DoTask messages.

Undo

```
procedure Undo;
void Undo (void);
```

Undo a task. The default method toggles the value of undone. Your subclass must store enough information to be able to undo an action. This is the method where you implement the undo.

Redo

```
procedure Redo;
void Redo (void);
```

Redo a task that was undone. The default method sends the task an Undo message. This method assumes that a redo is the same as undoing the undo. If your application implements **Redo** differently, you'll need to override this method.

Class resources

Resource	Description
STR# 130	List of strings for the wording of the **Undo/Redo** command. For instance, if you're implementing a "move" action, your string would be "Move". Each task contains the index of its string in this resource.

CTearChore 64

Introduction

CTearChore is a chore the notifies a tear-off menut that it has been torn off..

Heritage

Superclass	CChore
Subclasses	None

Using CTearChore

CTearChore is used in CTearOffMenu's `TornOff` method to let a menu that it has been torn off from the menu bar. `TornOff` creates a tear chore and assigns it as an urgent chore to the application. Your application should not have to use CTearChore directly, but you may find it useful as an example chore.

Variables

Variable	Type	Description
`itsTearOffMenu`	`CTearOffMenu`	The tear-off menu that has been torn off.

Methods

ITearChore

```
procedure ITearChore (aTearOffMenu: CTearOffMenu);
void ITearChore (CTearOffMenu *aTearOffMenu);
```

Create a tear chore. `ATearOffMenu` is stored in `aTearOffMenu`. CTearOffMenu's `TornOff` method creates a tear chore and assigns it as an urgent chore.

Perform

```
procedure Perform (var maxSleep: longint);
void Perform (long *maxSleep);
```

Sends a `MoveToCorner` method to `itsTearOffMenu`. This method does not change `maxSleep`.

CTearOffMenu 65

Introduction

CTearOffMenu is an abstract class that implements a Macintosh tear-off menu.

Heritage

Superclass	CDirector
Subclasses	None

Using CTearOffMenu

CTearOffMenu is a director that holds the pane of a menu that has been torn off from the menu bar. To use a CTearOffMenu, you need to create a subclass and override the initialization method so it creates a pane. The , create an MDEF class (CSelectorMDEF or another descendant of CPaneMDEF) and pass both the pane and the CTearOffMenu subclass to the initialization method.

The window that CTearOffMenu uses to display the tear-off menu is a floating window.

The Art Class example provided with the THINK Class Library uses two subclass of CTearOffMenu for the tool palette and the pattern palette.

Variables

Variable	Type	Description
itsPane	CPane	The pane being displayed in the menu
corner	Point	Top left of torn-off window.
margins	Rect	Space between window bounds and the pane in the window

Methods

ITearOffMenu

```
procedure ITearOffMenu (aSupervisor: CApplication;
   WINDid: Integer);
void ITearOffMenu (CApplication *aSupervisor,
   short WINDid);
```

Create a director for a tear-off menu. A tear-off menu's supervisor must be the application. `WINDid` is resource ID of the window that the tear-off menu appears in. Tear-off menus should use the Windoid `WDEF`. The source for this `WDEF` as well as the `WDEF` resource itself is in the `FW/Tearoffs` folder.

`ITearOffMenu` sets itsPane to `NIL`, so your subclass needs to create a pane and set itsPane to it. This should be the same pane that you pass to the CPaneMDEF's initialization method.

Suspend

```
procedure Suspend;
void Suspend(void);
```

Hides the tear-off menu when the application is suspended.

Resume

```
procedure Resume;
void Resume(void);
```

Shows the tear-off menu when the application is reactivated..

CloseWind

```
procedure CloseWind(theWindow: CWindow);
void CloseWind(CWindow *theWindow);
```

Close the tear-off menu. To close the tear-off menu, this method moves its window off the screen.

TornOff

```
procedure TornOff(aCorner: Point);
void TornOff(Point aCorner);
```

The menu has been torn off to the point `aCorner`. This method creates a tear chore (see CTearChore on page 441) that moves the window to the appropriate location..

MoveToCorner

```
procedure MoveToCorner;
void MoveToCorner(void);
```

Move the window to the corner stored by the `TornOff` method.

GetMacWindow

```
function GetMacWindow(void): WindowPtr;
WindowPtr GetMacWindow(void);
```

Return the window record for the tear-off menu window.

SetMargins

```
procedure SetMargins(aMargins: Rect);
void SetMargins(Rect *aMargins);
```

Set the margin between the window outline and the pane in the window. The margin is not a rectangle but the amount by which the pane's frame should be enlarged to create the gray outline of the tear-off menu.

GetMargins

```
procedure GetMargins(var theMargins: Rect);
void GetMargins(Rect *theMargins);
```

Return the margins in `theMargins`.

CTextEditTask 66

Introduction

CTextEditTask provides undo support for typing and editing in CAbstractText subclasses.

Heritage

Superclass	CTask
Subclasses	None

Using CTextEditTask

CTextEditTask implements undo for typing and the **Cut**, **Copy**, **Paste**, and **Clear** commands in CAbstractText subclasses. A text pane automatically creates an instance of this class when you type, press Backspace or Forward Delete, or choose an editing command.

You may need to create a subclass of CTextEditTask if you create your own subclass of CAbstractText, especially if your subclass allows text to have multiple styles or if it doesn't store its text in a single contiguous block of memory. For example, a CStyleText text pane uses a task of type CStyleTEEditTask that can deal with the style scrap format.

Variables

These are internal instance variables which only subclasses of CTextEditTask should use. In THINK C, they are protected.

Variable	Type	Description
itsTextPane	CAbstractText	Text pane that this task acts on.
editCmd	longint	Command being performed, cmdNull if typing
inserted	tTextRange	Info about the inserted text.
deleted	tTextRange	Info about the deleted text.
originalScrap	Handle	Contents of text scrap, before this task.
stillTyping	Boolean	TRUE if user is typing.
doText	Boolean	TRUE if this task changes the text in the text pane
doClip	Boolean	TRUE if this task changes the clipboard.
typingEvent	EventRecord	Event record for last keystroke.

Methods

Creation and destruction methods

ITextEditTask

```
procedure ITextEditTask (aTextPane: CAbstractText;
   anEditCmd: longint; firstTaskIndex: integer);
void ITextEditTask (CAbstractText *aTextPane,
   long anEditCmd, short firstTaskIndex);
```

Initialize this text edit task. anEditCmd is the command that this task is responding to, like cmdCut, cmdCopy, cmdPaste, or cmdClear. If the task is responding to typing, anEditCmd is cmdNull. ATextPane is the text pane for this task. FirstTaskIndex is the index of the first text edit Undo string in STR# 130. The text edit **Undo** strings are typically "Typing," "Cut," "Copy," "Paste," and "Clear." This method finds the index into STR# 130 for anEditCmd, sets doClip to TRUE if this command will change the contents of the clipboard, or cmdCut, sets doText to TRUE if this command adds or deletes text to the text pane, and saves the currently selected range.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of the memory for this object.

Accessing method

CanStillType

```
function CanStillType: Boolean;
Boolean CanStillType (void);
```

Returns TRUE if the user's typing doesn't start a new task. The user can type if the user hasn't tried to undo this task and stillTyping is TRUE.

Action methods

DoTask/Do

```
procedure Do;
void Do (void);
```

Perform an **Edit** menu command by sending the text pane a PerformEditCommand message. This method then stores the resulting selection

DoTyping

```
procedure DoTyping (theChar: char; keyCode: integer;
   macEvent: EventRecord);
void DoTyping (char theChar, short keyCode,
   EventRecord *macEvent);
```

Type a character. Depending on the character, this methods calls DoBackspace, DoFwdDelete or DoNormalChar. This method is called only when performing a command, not when undoing one.

Undo

```
procedure Undo;
void Undo (void);
```

Undo this task. This method saves the text the user inserted, removes the inserted text, and restores the text the user deleted. If you chose **Cut** or **Copy**, it also restores the old clipboard.

Redo

```
procedure Redo;
void Redo (void);
```

Redo this task, after it's been undone. This removes the text the user deleted and restores the text the user inserted. If you chose **Copy** or **Cut**, it restores the new clipboard.

CancelTyping

```
procedure CancelTyping;
void CancelTyping (void);
```

Stop accumulating characters that the user types. This method is called if this task was created to handle typing and you have just stopped to move the cursor or select **Undo**. After this method is called, you can undo the typing until you start editing again or initiate another task.

SelectionChanged

```
procedure SelectionChanged;
void SelectionChanged (void);
```

The selection has changed. This method sends a `CancelTyping` message.

Internal methods

DoNormalChar

```
procedure DoNormalChar (theChar: char);
void DoNormalChar (char theChar);
```

Handle a key that is not a Backspace or Forward Delete key, usually a cursor key or a character key. This method sends its text pane a `TypeChar` message.

DoBackspace

```
procedure DoBackspace;
void DoBackspace (void);
```

Handle the Backspace key. If you're performing this task, this method saves the character you're deleting.

DoFwdDelete

```
procedure DoFwdDelete;
void DoFwdDelete (void);
```

Handle the Forward Delete key. If you're performing this task, this method saves the character you're deleting.

SaveRange

```
procedure SaveRange (whichRange: tRangeSelector);
void SaveRange (tRangeSelector whichRange);
```

If `whichRange` is `kDeletedText`, save the text the user is about to delete. If `whichRange` is `kInsertedText`, save the text the user just inserted.

DeleteRange

```
procedure DeleteRange (whichRange: tRangeSelector);
void DeleteRange (tRangeSelector whichRange);
```

If `whichRange` is `kDeletedText`, delete the text the user deleted. If `whichRange` is `kInsertedText`, delete the text the user inserted.

RestoreRange

```
procedure RestoreRange (whichRange: tRangeSelector;
   killData: Boolean);
void RestoreRange (tRangeSelector whichRange,
   Boolean killData);
```

If `whichRange` is `kDeletedText`, restore the text the user deleted. If `whichRange` is `kInsertedText`, restore the text the user inserted. If `killData` is TRUE, this method disposes of this object's copy of the restored text.

StoreToClip

```
procedure StoreToClip (whichClip: tClipSelector);
void StoreToClip (tClipSelector whichClip);
```

If `whichClip` is `kOldClip`, store the original scrap text. If `whichClip` is `kNewClip`, store the deleted text to the clipboard.

CTextEnvirons 67

Introduction

CTextEnvirons maintains a Quickdraw text drawing environment for any pane.

Heritage

Superclass	CEnvironment
Subclasses	None

Using CTextEnvirons

Every pane has an `itsEnvironment` instance variable. If this variable points to a descendant of CEnvironment, the `Prepare` method sends it a `Restore` message to set up the drawing environment for the pane.

You can use CTextEnvirons to make sure that a pane's text drawing characteristics are set up correctly. CTextEnvirons maintains the font, the size of the font, the font style, and the drawing transfer mode.

Suppose you have a pane that lets the user set the font and size of a text display. When you create your pane, you read the settings into a `TextInfoRec`, create a `CTextEnvirons` object, then set the `itsEnvironment` instance variable to point to it. Whenever the pane needs to be drawn, the `Prepare` method sends it a `Restore` message so the drawing mode is set up correctly.

Here's how you might set up a CTextEnvirons object for a pane in Pascal:

```
procedure CSomeDisplayPane.ISomeDisplayPane (
        anEnclosure: CView;
        aSupervisor: CBureaucrat);
   var
      aTextInfo: TextInfoRec;
begin
   ...
   aTextInfo.fontNumber := ReadStoredFont;
   aTextInfo.theSize := ReadStoredSize;
   aTextInfo.theStyle := [];
   aTextInfo.theMode := srcCopy;
   new(CTextEnvirons(itsEnvironment));
   CTextEnvirons(itsEnvironment).SetTextInfo
                                    (aTextInfo);
   ...
end;
```

Here's how you might do the same thing in C:

```
void CSomeDisplayPane::ISomeDisplayPane (
        CView *anEnclosure,
        CBureaucrat *aSupervisor)
{
   TextInfoRec aTextInfo;
   ...
   aTextInfo.fontNumber = ReadStoredFont();
   aTextInfo.theSize = ReadStoredSize();
   aTextInfo.theStyle = 0;
   aTextInfo.theMode = srcCopy;
   itsEnvironment = new CTextEnvirons;
   ((CTextEnvirons *) itsEnvironment)->
                     SetTextInfo(&aTextInfo);
   ...
}
```

Variables

Variable	Type	Description
textInfo	TextInfoRec	Text characteristics

The TextInfoRec looks like this:

Field	Type	Description
fontNumber	Integer	The number of the font
theSize	Integer	Size of the font
theStyle	Style	Style of the font (in THINK C, this type is short)
theMode	Integer	Text transfer mode

Methods

ITextEnvirons

```
procedure ITextEnvirons;
void ITextEnvirons (void);
```

Initialize every field of the textInfo record to zero. These settings correspond to the default system font, the default system size, plain style, and the scrCopy transfer mode.

Restore

```
procedure Restore;
void Restore (void);
```

Sets the Quickdraw text drawing characteristics to the values previously stored with SetTextInfo. This method uses the standard QuickDraw text setting routines: TextFont, TextSize, TextFace, and TextMode. This method also calls the Toolbox routine PenNormal.

SetTextInfo

```
procedure SetTextInfo (aTextInfo: TextInfoRec);
void SetTextInfo (TextInfoRec *aTextInfo);
```

Sets the text drawing characteristics to the values in aTextInfo. The next time the pane is redrawn, the QuickDraw text drawing characteristics will be set to the values supplied.

GetTextInfo

```
procedure GetTextInfo (var aTextInfo: TextInfoRec);
void GetTextInfo (TextInfoRec *aTextInfo);;
```

Get the current text drawing characteristics from the object.

CTextStyleTask 68

Introduction

CTextStyleTask provides undo support for style commands in CAbstractText subclasses.

Heritage

Superclass	CTask
Subclasses	CStyleTEStyleTask

Using CTextStyleTask

CTextStyleTask provides undo support for font, size, style, alignment, and spacing commands in CAbstractText subclasses. A text pane automatically creates an instance of this class when you choose a style command.

You may need to create a subclass of CTextStyleTask if you create your own subclass of CAbstractText, especially if your subclass allows text to have multiple styles or if it doesn't store its text in a single contiguous block of memory. For example, a CStyleText text pane uses a task of type CStyleTEStyleTask that can deal with the style scrap format.

Variables

These are internal instance variables which only subclasses of CTextStyleTask should use. In THINK C, they are protected.

Variable	Type	Description
itsTextPane	CAbstractText	Text pane that this task acts on.
oldStyle	TextStyle	Style before this task.
oldAlignCmd	longint	Alignment before this task.
oldSpacingCmd	longint	Spacing before this task.
styleCmd	longint	Command this task performs.
styleAttribute	integer	Style attributes affected by this task.

Methods

Construction method

ITextEditTask

```
procedure ITextStyleTask (aTextPane: CAbstractText;
   aStyleCmd: longint; taskIndex: integer);
void ITextStyleTask (CAbstractText *aTextPane,
   long aStyleCmd, short taskIndex);
```

Initialize this text style task. AStyleCmd is the command that this task is responding to. ATextPane is the text pane for this task. TaskIndex is the index of this command's **Undo** string in STR# 130.

Action methods

DoTask/Do

```
procedure DoTask;
void Do (void);
```

Save the original formatting, then performs the user's formatting command.

Undo

```
procedure Undo;
void Undo (void);
```

Save the current formatting and restore the previously saved formatting. This method handles both **Undo** and **Redo**.

Internal methods

SaveStyle

```
procedure SaveStyle;
void SaveStyle (void);
```

Save the style of the text. Depending on the text pane, this method saves either the style for the whole text pane (like CEditText) or the style for the current selection (like CStyleText).

RestoreStyle

```
procedure RestoreStyle;
void RestoreStyle (void);
```

Restores the previously saved formatting.

CView 69

Introduction

CView is an abstract class for implementing objects that have a visual representation. Every object in the visual hierarchy is a descendant of this class.

Heritage

Superclass	CBureaucrat
Subclasses	CDesktop
	CPane
	CWindow

Using CView

CView is an abstract class for implementing objects with a visual representation. In other words, anything you can see on the screen is a descendant of CView. Views respond to visual commands involving the mouse. And because CView is a descendant of CBureaucrat, a view can be one of the links in the chain of command.

The standard classes define three subclasses of CView. These are the desktop, windows, and panes. Most of the time, you'll be dealing with panes. As you work with panes and descendants of CPane, keep in mind that all methods that apply to views apply to them as well.

Views and the visual hierarchy

All views have an enclosure that specifies its place in the visual hierarchy. Each view can enclose several subviews. The top of the visual hierarchy, the desktop, is the only view that does not have an enclosure. The desktop encloses all the windows in your application. Each window encloses one or mores panes. Panes can enclose other panes.

The desktop handles some visual commands, like mouse clicks, and sends them on to the appropriate window. The switchboard sends window related

messages, like updates and activates, directly to a window which sends it to its subviews.

By default, a view does not process mouse clicks. If a view can respond to mouse clicks, you need to call `SetWantsClicks(TRUE)` when you create it to let `DispatchClick` and `AdjustCursor` know that they should look for clicks in mouse movement in that view.

Views and the chain of command

Each view has a supervisor which is the view's boss in the chain of command. The desktop's supervisor is always the application. A window's supervisor is always its director or document. A pane's supervisor is usually its director or document.

The desktop and windows are almost never the first in the chain of command. In other words, they're almost never the gopher. Panes, on the other hand, are frequently made the gopher. For instance, you need to make an edit pane the gopher so it can respond to typing and menu commands.

If a view can be a gopher, you need to call `SetCanBeGopher(TRUE)` so `DispatchClick` can make the view the gopher and let the previous gopher know that it's not the gopher anymore. Although you can force a view to be the gopher by setting the `gGopher` instance variable, you should rely on the gopher-setting mechanism of `BecomeGopher`.

Using Balloon Help with views

Any view can have a help balloon associated with it. The THINK Class Library uses 'hrct' resources to specify help balloons. The Macintosh Help Manager, described in *Inside Macintosh VI*, uses a combination of `'hwin'` and `'hrct'` resources to display help balloons for stationary windows. The THINK Class Library uses only `'hrct'` resource. They should not be associated with `'hwin'` resources.

Each window holds a resource ID to an `'hrct'` resource for help associated with that window. If the window doesn't provide a resource ID for the `'hrct'` resource, it uses the default `'hrct'` resource whose ID is `kDefaultHelpResID` (128).

If you want to provide help for a pane, set the `helpResIndex` instance variable, which it inherits from CView, as an index into the `'hrct'` resource. Otherwise, just set `helpResIndex` to 0.

The `DispatchCursor` method uses `GetBalloonInfo` and `ShowHelpBalloon` to look for the help resource and to display them.

Variables

Variable	Type	Description
`macPort`	`GrafPtr`	Mac drawing port for the view
`itsEnclosure`	`CView`	View which totally encloses this one
`itsSubviews`	`CList`	Views contained within this view
`visible`	`Boolean`	Is the view visible?
`active`	`Boolean`	Is the view active?
`wantsClicks`	`Boolean`	Does the view handle mouse clicks?
`canBeGopher`	`Boolean`	Can this view be the gopher?
`ID`	`longint`	The identifier for this view.
`usingLongCoord`	`Boolean`	TRUE if using 32-bit coordinates
`helpResIndex`	`integer`	Index into `'hrct'` resource for balloon help

The following three variables are class variables. In Pascal these are global variables. CView uses these class variables for tracking help balloons and for optimizing calls to `Prepare`. In general, you won't need to use them.

Variable	Type	Description
`cCurrHelpView`	`CView`	Used in `DispatchCursor` to determine whether a help balloon was displayed
`cLastHelpView`	`CView`	The view that is showing a help balloon
`cPreparedView`	`CView`	Currently prepared view

Methods

Construction and destruction methods

IView

```
procedure IView (anEnclosure: CView;
   aSupervisor: CBureaucrat);

void IView (CView *anEnclosure,
   CBureaucrat *aSupervisor);
```

Initialize a view. Views start out with no port, no subviews, invisible, and inactive. By default, views don't want clicks. AnEnclosure is the view that

completely encloses this view. ASupervisor is the bureaucrat that gets command messages for the commands this view can't handle.

IViewRes

```
procedure IViewRes (rType: ResType; resID: integer;
   anEnclosure: CView; aSupervisor: CBureaucrat);
void IViewRes (ResType rType, short resID,
   CView *anEnclosure, CBureaucrat *aSupervisor);
```

Initialize a view from a resource template. Each subclass of CView overrides this method to use its own resource template. `RType` is the resource type for the CView subclass you want to initialize. `ResID` is the resource ID of the resource. `AnEnclosure` is the view that completely encloses this view. `ASupervisor` is the bureaucrat that gets command messages when the view can't handle them.

To initialize a view from a resource file, use a `'View'` resource.

IViewTemp

```
procedure IViewTemp (anEnclosure: CView;
   aSupervisor: CBureaucrat; viewData: Ptr);
void IViewTemp (CView *anEnclosure,
   CBureaucrat *aSupervisor, Ptr viewData);
```

The `IViewRes` method sends an `IViewTemp` message to initialize a view from a resource template. All subclasses of CView override this method so they can be initialized from resource templates.

Free/Dispose

```
procedure Free;
void Dispose (void);
```

Dispose of a view. Disposes of all subviews

Accessing methods

IsVisible

```
function IsVisible: Boolean;
Boolean IsVisible (void);
```

Return TRUE if the view is visible.

IsActive

```
function IsActive: Boolean;
Boolean IsActive (void);
```

Return TRUE if the view is active.

ReallyVisible

```
function ReallyVisible: Boolean;
Boolean ReallyVisible (void);
```

Return TRUE if the view is visible and if its enclosure is `ReallyVisible`.

GetMacPort

```
function GetMacPort: GrafPtr;
GrafPtr GetMacPort (void);
```

Get the `GrafPort` associated with this view.

GetOrigin

```
procedure GetOrigin (var theHOrigin,
   theVOrigin: longint);
void GetOrigin (long *theHOrigin, long *theVOrigin);
```

Get the origin of the view (the top left corner). The default method returns (0, 0).

GetFrame

```
procedure GetFrame (var theFrame: LongRect);
void GetFrame (LongRect *theFrame);
```

Get the frame of the view. The default method does nothing. View subclasses (like Pane) must override this message.

GetInterior

```
procedure GetInterior (var theInterior: LongRect);
void GetInterior (LongRect *theInterior);
```

Get the interior of the view. The default method returns what `GetFrame` returns. If the interior of a particular subclass is not the same as the frame, the class must override this method.

GetAperture

```
procedure GetAperture (var theAperture: LongRect);
void GetAperture (LongRect *theAperture);
```

Get the aperture of the view. (The aperture is the visible portion of a view.) The default method does nothing. Subclasses must override this method.

Contains

```
function Contains (thePoint: Point): Boolean;
Boolean Contains (Point thePoint);
```

Return TRUE if the view contains `thePoint`. The default method always returns FALSE.

SetWantsClicks

```
procedure SetWantsClicks (aWantsClicks: Boolean);
void SetWantsClicks (Boolean aWantsClicks);
```

If `aWantsClicks` is `true`, the view will report clicks in itself. By default, views don't want clicks.

GetWantsClicks

```
function GetWantsClicks: Boolean;
Boolean GetWantsClicks (void);
```

Returns TRUE if this view wants to receive clicks

SetCanBeGopher

```
procedure SetCanBeGopher (fCanBeGopher: Boolean);
void SetCanBeGopher (Boolean fCanBeGopher)
```

If `fCanBeGopher` is TRUE, this view can become the gopher. By default, a view can't become the gopher.

CanBeGopher

```
function CanBeGopher: Boolean;
Boolean CanBeGopher (void)
```

Returns TRUE if this view can become the gopher.

SetID

```
procedure SetID (anIdentifier: longint);
void SetID (long anIdentifier)
```

Sets this view's ID to be `anIdentifier`. By default, the ID is 0 (zero). It is up to you to provide IDs and to guarantee that they're unique. View IDs give you a way to identify particular views within the view hierarchy. You can use Macintosh style identifiers like 'MyVu' or plain numeric constants.

GetID

```
function GetID: long;
long GetID (void)
```

Returns the ID for this view.

UseLongCoordinates

```
procedure UseLongCoordinates (fUsing: Boolean);
void UseLongCoordinates (Boolean fUsing);
```

Specify whether the view should use 32-bit coordinates (long coordinates) or 16-bit QuickDraw coordinates. If `fUsing` is TRUE, the view is using 32-bit coordinates.

Appearance methods

Show

```
procedure Show;
void Show (void);
```

Make a view visible.

Hide

```
procedure Hide;
void Hide (void);
```

Hide a view. If the view is the gopher, make its supervisor the gopher.

Activate

```
procedure Activate;
void Activate (void);
```

Make a view active. Activate all the subviews as well.

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

Make a view inactive. Deactivate all the subviews as well. If the view is the gopher, make its supervisor the gopher.

Mouse methods

DispatchClick

```
procedure DispatchClick (var macEvent: EventRecord);
void DispatchClick (EventRecord *macEvent);
```

Find out which subview got the click and send it a `DispatchClick` message. If there are no subviews, the click is for this view. If the view can be the gopher, `DispatchClick` calls `BecomeGopher(TRUE)` to make the view the gopher. Then `DispatchClick` sets calls `Prepare` to set up the drawing environment. Finally, `DispatchClick` sends a `DoClick` message to the view to handle the click. Ordinarily, you should not need to override this method.

DoClick

```
procedure DoClick (hitPt: Point;
   modifierKeys: integer; when: longint);
void DoClick (Point hitPt, short modifierKeys,
   long when);
```

The mouse went down in this view. If the view is not using long coordinates, `hitPt` is in frame coordinates. If the view is using long coordinates, `hitPt` is given in QuickDraw coordinates. You can use CPane's `QDToFrame`, `QDToFrameR` to convert from long coordinates to frame coordinates, and `FrameToQD` and `FrameToQDR` to convert from long coordinates to QuickDraw coordinates. Subclasses must override this method.

These conversion routines are described on page 359

HitSamePart

```
function HitSamePart (var pointA, pointB: Point):
   Boolean;
Boolean HitSamePart (Point pointA, Point pointB);
```

Check whether two points hit the same part of the view. The default method always returns `TRUE`. Subclasses that override this method should decide what constitutes a part and how close is close.

DoMouseUp

```
procedure DoMouseUp (macEvent: EventRecordPtr);
void DoMouseUp (EventRecord *macEvent);
```

The mouse went up in a view. Default method does nothing.

Cursor methods

DispatchCursor

```
procedure DispatchCursor (where: Point;
   mouseRgn: RgnHandle);
void DispatchCursor (Point where, RgnHandle mouseRgn);
```

Find which view the cursor is in. If the cursor is in this view, send it an `AdjustCursor` message. If the cursor is in a subview, send it a `DispatchCursor` message. `DispatchCursor` sends `AdjustCursor` and `DispatchCursor` messages only to views that want clicks.

If the balloon help system is available, and the application has help resources, and no other view has already displayed the help text, `DispatchCursor` display the help balloon associated with this view.

Your application should not need to override this class.

AdjustCursor

```
procedure AdjustCursor (where: Point;
   mouseRgn: RgnHandle);
void AdjustCursor (Point where, RgnHandle mouseRgn);
```

Adjust the cursor. A view gets an `AdjustCursor` message when the cursor moves into it and if the view wants clicks. The default method sets the cursor to an arrow. Your subclass should override this method to set the cursor to whatever is appropriate for the view. See the `AdjustCursor` method for CAbstractText for an example.

If you use only one cursor within a view, you can ignore the two parameters to this method. If you want to use different cursor shapes within a pane, you need use these two parameters.

It's unlikely that you'll want multiple cursors for a view that's not a pane. The CPane class inherits this method.

The `where` parameter tells you where the cursor is in window coordinates. You can use the `WindToFrame` pane method to convert it to frame coordinates. The `mouseRgn` parameter is the region in which the cursor shape stays the same. In other words, your pane will not get an `AdjustCursor` message again until the cursor leaves this region. The `mouseRgn` is specified in global coordinates.

Suppose you're displaying a map of the United States in a pane, and you want the cursor to be a star when it's over Texas. You use the `where` parameter, converted from window coordinates to frame coordinates, to determine that you're in Texas, and you change the cursor to a star.

Now, when you leave Texas, you want to set the cursor shape to something else. Since your pane gets `AdjustCursor` messages only when the cursor leaves the `mouseRgn`, you have to change the `mouseRgn`. You create a region with the shape of the state of Texas, convert this region to global coordinates, and make it the `mouseRgn`.

Note

The mouse region should be the intersection of the original mouse region and the one you're specifying. This way, you preserve any clipping boundaries that the original mouse region had accounted for.

This is what the `AdjustCursor` method for this example might look like in Pascal:

```
procedure MapPane.AdjustCursor (where: Point;
   mouseRgn: RgnHandle);
var
   locWhere: Point;
   TexasRgn: RgnHandle;
   TexasRect: Rect;
   tempRect: Rect;

begin
   TexasRgn := NewRgn;
   locWhere := where;
   WindToFrame(locWhere);

      { If we're not in Texas, }
      { use the default.       }
   if (not PtInTexas(locWhere)) then
     begin
      inherited AdjustCursor(where, mouseRgn);
      exit(AdjustCursor);
     end

      { Set the star cursor. }
   SetCursor(star);

      { Calculate the mouse region }
      { in frame coords.           }
   OpenRgn;
   DrawTexas;
   CloseRgn(TexasRgn);
```

```
        { Convert it to global coords. }
      TexasRect := TexasRgn^^.rgnBBox;
      tempRect := TexasRect;
      FrameToGlobalR(tempRect);
      OffsetRgn(TexasRgn,
         tempRect.left - TexasRect.left
         tempRect.top - TexasRect.top);

         { Set the mouse region. }
      SectRgn(mouseRgn, TexasRgn, mouseRgn);
   end;
```

And this is what the `AdjustCursor` method might look like in C:

```
void MapPane::AdjustCursor(Point where,
   RgnHandle mouseRgn);
{
   Point       locWhere;
   RgnHandle   TexaxRgn = NewRgn();
   Rect        TexasRect;
   Rect        TempRect;

   locWhere = where;
   WindToFrame(&locWhere);

      /* If we're not in Texas, */
      /* use the default.       */
   if (!ptInTexas(locWhere)) {
      inherited::AdjustCursor(where, mouseRgn);
      return ;
   }

      /* Set the star cursor. */
   SetCursor(star);

      /* Calculate the mouse region */
      /* in frame coords.           */
   OpenRgn();
   DrawTexas();
   CloseRgn(TexasRgn);

      /* Convert it to global coords. */
   TexasRect = (**TexasRgn).rgnBBox;
   tmpRect = TexasRect;
   FrameToGlobalR(&tempRect);
   OffsetRgn(TexasRgn,
      tempRect.left - TexasRect.left,
      tempRect.top - TexasRect.top)

      /* Set the mouse region. */
   SectRgn(mouseRgn, TexasRgn, mouseRgn);
}
```


GetBalloonInfo

```
procedure GetBalloonInfo (
   var helpData: HMMessageRecord; var tip: Point;
   var altertnateRect: Rect; var tipProc: Ptr;
   var variant: integer; var method: integer);
void GetBalloonInfo (HMMessageRecord *helpData,
   Point *tip, Rect *alternateRec, Ptr *tipProc,
   short *theProc, short *variant, short *method);
```

Set up the parameters needed to show a help balloon. If `helpResIndex` is greater than 0, use it as an index into the `'hrct'` resource associated with this view. If the view is a pane, the enclosing window holds the resource id of the `'hrct'` resource, otherwise it uses `'hrct'` with the ID `kDefaultHelpResID` (128).

This method calls the Help Manager's `HMGetIndHelpMsg` routine to get the information from the `'hrct'` resource.

ShowHelpBalloon

```
procedure ShowHelpBalloon (helpData: HMMessageRecord;
   tip: Point; altertnateRect: Rect; tipProc: Ptr;
   variant: integer; method: integer);
void ShowHelpBalloon (HMMessageRecord *helpData,
   Point tip, Rect *altRec, Ptr tipProc,
   short theProc, short variant, short method);
```

This method uses the data from `GetBalloonInfo` and calls the Help Manager routine `HMShowBalloon` to display the help balloon. If there is an error, this method calls `Failure`. `HelpData` comes from the `'hrct'` resource. `Tip`, as returned from `GetBalloonInfo` is set to the center of the view, in global coordinates. `AltRect` is the aperture of the view. `TipProc` is NIL. `TheProc` is 0, the standard definition function. Variant and method are both read from the `'hrct'` resource.

GetHelpResID

```
function GetHelpResID : integer;
short GetHelpResID (void);
```

Return the resource ID of the `'hrct'` resource that has the balloon help information for this view. This method returns the ID of the TCL's default `'hrc'` resource: `kDefaultResID`.

Subview Management methods

AddSubview

```
procedure AddSubview (theSubview: CView);
void AddSubview (CView *theSubview);
```

Add `theSubview` to the view's `itsSubviews` list. You should not use or override this method.

RemoveSubview

```
procedure RemoveSubview (theSubview: CView);
void RemoveSubview (CView *theSubview);
```

Remove `theSubview` from the `itsSubviews` list.

FindSubview

```
function FindSubview (hitPt: Point): CView;
CView* FindSubview (Point hitPt);
```

Find the subview that wants clicks and contains `hitPt`. `HitPt` is in window coordinates. This method finds the topmost subview. You should not override this method.

FindViewByID

```
function FindViewByID (anID: longint): CView;
CView* FindViewByID (long anID);
```

Find the subview whose ID is `anID`. If more than one view has the same ID, this method finds the topmost subview.

MatchView

```
function MatchView (function matchProc(aView: CView;
   matchData: Ptr): Boolean;
   matchData: Ptr): CView;
CView* MatchView (MatchViewProc matchProc,
   void *matchData)
```

Find the first subview that returns TRUE for `matchProc`. This method finds the topmost subview.

`MatchProc` is a pointer to function of the form:

```
function matchProc (aView: CView;
   matchData: Ptr);
```

or in C:

```
Boolean matchProc (CView *aView,
   void *matchData);
```

`MatchData` can be anything that your `matchProc` needs. For an example of `MatchView`, see the definition of `FindViewByID`.

SubpaneLocation

```
procedure SubpaneLocation (hEncl, vEncl: longint;
   var hLocation, vLocation: longint);
void SubpaneLocation (long hEncl, long vEncl,
   long *hLocation, long *vLocation);
```

Return the position of a subview at `hEncl`, `vEncl` in `hLocation` and `vLocation` in window coordinates.

Prepare

```
procedure Prepare;
void Prepare (void);
```

Prepare to draw or to do something after a click. The default method sets the class variable `cPreparedView` to `this`. The class variable `cPreparedView` points to the most recently prepared view. CPane uses `cPreparedView` to avoid redundant calls to `Prepare`.

View classes must override this method. The overriden methods should set `cPreparedView` to `this` or call the inherited method. See the definition of `Prepare` in CPane for an example.

ForceNextPrepare

```
procedure ForceNextPrepare;
void ForceNextPrepare (void);
```

Clear `cPreparedView` to disable the optimization on the next call to `Prepare`. If you change the port, the clipping region, or the origin of a view, you must call this method to disable the `Prepare` optimization.

FrameToGlobalR

```
procedure FrameToGlobalR (frameRect: LongRect;
   var globalRect: Rect);
void FrameToGlobalR (LongRect *frameRect,
   Rect *globalRect);
```

Convert `frameRect` from frame coordinates to global coordinates and put the converted coordinates into `globalRect`. The default method does nothing. View subclasses must override thismethod.

Class resources

The `IViewRes` method lets you initialize any descendant of CView with a resource template.

Resource	Class
`Bord`	CBorder
`Pane`	CPane
`Pano`	CPanorama
`PctP`	CPicture
`ScPn`	CScollPane
`StTx`	CStaticText, CEditText
`View`	CView

You can use ResEdit to create the resource templates for each class.

Note

Your package includes a file called TCL TMPLs that contains a set of TMPL resources you can install into ResEdit. These TMPLs let you create and edit the resource above easily. See the instructions in "Installing the TMPL Resources into ResEdit" on page 13 to learn how to install these TMPLs into ResEdit.

CWindow 70

Introduction

CWindow implements a class necessary to manipulate a Macintosh window.

Heritage

Superclass	CView
Subclasses	None

Using CWindow

Virtually every application you write on the Macintosh uses windows. The CWindow class implements methods to manipulate Macintosh windows. In the THINK Class Library, window objects are merely visual entities. They're not designed to interact directly with the application. The class CDirector manages communication between a window and the application. The CDocument class, a subclass of CDirector, manages communication between the application, a window, and a file.

Objects of class CWindow usually belong to directors. Under normal circumstances, you should not need to define a subclass of CWindow or to manipulate one directly. The only thing you really need to do to a window is create it and set its options.

You can specify that a window be modal. If a modal window is the frontmost window, CDesktop's DispatchClick method will not let a click in another window deactivate the modal window, though mouse clicks in the menu bar are handled. Be sure that your program provides a way to close modal windows.

The Macintosh windows associated with CWindow objects have a window kind of OBJ_WINDOW_KIND. If your application uses windows that are not associated with a CWindow object— from a utility library, for instance— you should override the CDesktop methods DispatchClick and DispatchCursor and the CSwitchboard methods DoUpdate,

DoActivate, and DoDeactivate to check the window kind of the Macintosh windows they deal with.

Variables

The enclosure for all windows must be gDesktop.

Global variable

Variable	Type	Description
gDesktop	CDesktop	The global desktop which acts as the top of the visual hierarchy and the enclosure for all windows.

Instance variables

Variable	Type	Description
procID	integer	Window definition ID used to create the window.
sizeRect	Rect	Minimum and maximum size of the window
floating	Boolean	Is this a floating window?
isColor	Boolean	Is this a color window?
isModal	Boolean	Is it currently modal?
actClick	Boolean	Process mouse click which activates window?
location	Point	Current window location, used when "hiding" a window while suspended
helpResID	integer	Resource ID of 'hrct' help resource for panes in this window.

Methods

Construction and destruction methods

IWindow

```
procedure IWindow (WINDid: integer;
   aFloating: Boolean; anEnclosure: CDesktop;
   aSupervisor: CDirector);
void IWindow (short WINDid, Boolean aFloating,
   CDesktop *anEnclosure,
   CDirector *aSupervisor);
```

Initialize a window object from a WIND resource. If Color QuickDraw is available, this method creates a color window.

WINDid is the id of the WIND resource. If aFloating is true, the window is a floating window. In this case, anEnclosure must refer to a floating window desktop.

AnEnclosure is the view the window belongs to. A window's enclosure should always be gDesktop. ASupervisor is the window's supervisor. Usually a window's supervisor is the director it belongs to. IWindow sends the window a MakeMacWindow message to actually allocate space in memory for the window.

By default, a window is not modal. To make a window modal, use the SetModal method onpage 479.

By default, helpResID is set to kDefaultResID. HelpResID is the resource ID of the 'hrct' resource associated with this window. To change it use the SetHelpResID method on page 480. For more information about using Balloon Help with panes, see "Using Balloon Help with views" on page 462.

INewWindow

```
procedure INewWindow (bounds: Rect;
   fVisible: Boolean; aProcID: integer;
   fFloating: Boolean; Boolean fHasGoAway
   anEnclosure: CDesktop; aSupervisor: CDirector);
void INewWindow (Rect *bounds,
   Boolean fVisible, short aProcID,
   Boolean fFloating, Boolean fHasGoAway,
   CDesktop *anEnclosure,
   CDirector *aSupervisor);
```

Initialize a window from the parameters in the argument list. If Color QuickDraw is available, this method creates a color window.

Bounds is a rectangle that describes the size and position of the window. AProcID is an integer that describes that type of the window.

If fVisible is TRUE, the window is drawn immediately after it's created. If fHasGoAway is TRUE, the window has a close box in.

The other parameters are identical to IWindow's parameters above.

Free/Dispose

```
procedure Free;
void Dispose(void);
```

Dispose of the window and all of its subpanes. This method also removes the window from the desktop.

MakeMacWindow

```
procedure MakeMacWindow (WINDid: integer);
void MakeMacWindow (short WINDid);
```

Create a Macintosh window from a `WIND` resource. This method lets the Window Manager allocate memory itself. If you want to create your windows differently, override this method. Bear in mind that floating windows should be in front (-1) and that non-floating windows should be behind.

MakeNewMacWindow

```
procedure MakeNewMacWindow (bounds: Rect;
   aProcID: integer; fHasGoAway: Boolean);
void MakeNewMacWindow (Rect *bounds, short aProcID,
   Boolean fHasGoAway)
```

Create a Macintosh window from the parameters in the argument list. `Bounds` is a rectangle that describes the size and position of the window. `AProcID` is an integer that describes that type of the window. If `FHasGoAway` is TRUE, the window will have a close box in the left side of the title bar.

This method lets the Window Manager allocate memory itself. If you want to create your own windows differently, override this method. For example, you might want to perform your own memory allocation or to create color windows. For compatibility with the Desktop class, the convention of putting floating windows in front and non-floating windows in back should be observed. Bear in mind that floating windows should be in front (-1) and that non-floating windows should be behind.

Close

```
procedure Close;
void Close (void);
```

Close a window. A window object gets this message as a result of a click in the close box. The default method sends the window's supervisor (a director) a `CloseWind` message.

Accessing methods

GetFrame

```
procedure GetFrame (var theFrame: LongRect);
void GetFrame(LongRect *theFrame);
```

Get the frame of a window. The frame of the window includes the one pixel border around the window. The top, left of `theFrame` is always (-1,-1)

GetInterior

```
procedure GetInterior (var theInterior: LongRect);
void GetInterior(LongRect *theInterior);
```

Get the interior of a window. The interior of the window is the same as its `portRect` and does not include any of the border pixels. The top left of `theInterior` is always (0, 0).

GetAperture

```
procedure GetAperture (var theAperture: LongRect);
void GetAperture (LongRect *theAperture);
```

Return the drawable area of the window. For windows, this is the entire window. The top, left of `theAperture` is always (0,0).

IsFloating

```
function IsFloating: Boolean;
Boolean IsFloating (void);
```

Return TRUE if this is a floating window.

IsModal

```
function IsModal: Boolean;
Boolean IsModal (void);
```

Return TRUE if this is a modal window.

SetModal

```
procedure SetModal (fModal: Boolean);
void SetModal (Boolean fModal);
```

Specify whether a window is modal. If `fModal` is TRUE, the window is modal. If `fModal` is FALSE, it is not modal. If a modal window is the frontmost window, mouse clicks anywhere except in the window and in the menu bar are ignored.

IsColor

```
function IsColor: Boolean;
Boolean IsColor (void);
```

Return TRUE if this is a color window.

SetTitle

```
procedure SetTitle (theTitle: Str255);
void SetTitle (Str255 theTitle);
```

Set the window's title.

GetTitle

```
procedure GetTitle (var theTitle: Str255);
void GetTitle (Str255 theTitle);
```

Get the window's title.

SetActClick

```
procedure SetActClick (anActClick: Boolean);
void SetActClick (Boolean anActClick);
```

If `anActClick` is TRUE, the window processes the mouse click that activated it.

WantsActClick

```
function WantsActClick: Boolean;
Boolean WantsActClick (void);
```

Return TRUE if the window processes the mouse click that activates it.

Contains

```
function Contains (thePoint: Point): Boolean;
Boolean Contains (Point thePoint);
```

Return TRUE if thePoint is inside the window. ThePoint is in global coordinates.

SetSizeRect

```
procedure SetSizeRect (aSizeRect: Rect);
void SetSizeRect (Rect *aSizeRect);
```

Set the minimum and maximum size for the window. ASizeRect's top,left are the minimum height and width. ASizeRect's bottom,right are the maximum.

SetStdState

```
procedure SetStdState (aStdState: Rect);
void SetStdState (Rect *aStdState);
```

Set the standard state of a window. The standard state is the size and location of the window when it's zoomed out. Before you send this message to a window, make sure that the Macintosh window associated with it supports zooming. If it does, the spareFlag in the window record is true.

SetHelpResID

```
procedure SetHelpResID (aResID: integer);
void SetHelpResID (short aResID);
```

Set the resource ID of the 'hrct' Balloon Help resource associated with this window. For more information about using Balloon Help with the THINK Class Library, see "Using Balloon Help with views" on page 454.

GetHelpResID

```
function GetHelpResID: integer;
short GetHelpResID (void);
```

Get the resource ID of the 'hrct' resource associated with this window. If you do not set one with SetHelpResID, helpResID is set to kDefaultHelpResID by default.

Appearance methods

Show

```
procedure Show;
void Show (void);
```

Show a window. Also sends its enclosure (the desktop) a ShowWind message.

Hide

```
procedure Hide;
void Hide (void);
```

Hide a window. Also sends its enclosure (the desktop) a HideWind message.

Activate

```
procedure Activate;
void Activate (void);
```

Activate a window and all of its panes. Also sends its supervisor (a director) an ActivateWind message.

Deactivate

```
procedure Deactivate;
void Deactivate (void);
```

Deactivate a window and all of its panes. Also sends its supervisor (a director) a DeactivateWind message.

Select

```
procedure Select;
void Select (void);
```

Select a window by bringing it to the front and making it active. Sends its enclosure (the desktop) a SelectWind message.

ShowResume

```
procedure ShowResume;
void ShowResume (void);
```

Make a window visible when an application resumes. If you hide your windows when your application is suspended, send it this message in its director's Resume method.

HideSuspend

```
procedure HideSuspend;
void HideSuspend (void);
```

Hide a window when an application is suspended. To hide a window when the application is being suspended, send the window this message in its director's Suspend method. The HideSuspend method doesn't actually hide the window. Instead, it moves it out of the visible range so the front-to-back ordering of the windows doesn't change.

ShowOrHide

```
procedure ShowOrHide (showFlag: Boolean);
void ShowOrHide (Boolean showFlag);
```

Make a window visible or invisible without generating any update or activate events.

Size and location methods

Drag

```
procedure Drag (macEvent: EventRecord);
void Drag (EventRecord *macEvent);
```

Drag a window. This method is invoked when the user clicks in a window's drag region. The default method sends a DragWind message to the desktop.

Resize

```
procedure Resize (macEvent: EventRecord);
void Resize (EventRecord *macEvent);
```

Resize a window. This method is invoked when you click and drag the window's grow region. After resizing, this method sends a `ChangeSize` message to the window.

Zoom

```
procedure Zoom (direction: integer);
void Zoom (short direction);
```

Zoom a window. This method is invoked when the user clicks in a window's zoom box. Direction is either `inZoomIn` or `inZoomOut`. After changing the size of the window, this method sends an `AdjustToEnclosure` message to its panes.

Move

```
procedure Move (hGlobal, vGlobal: integer);
void Move (short hGlobal, short vGlobal);
```

Move the window to the specified location in global coordinates. Note that the new position refers to the top left corner of the window's content region. The title bar will be above the specified coordinates.

ChangeSize

```
procedure ChangeSize (width, height: integer);
void ChangeSize (short width, short height);
```

Change the size of the window to the specified width and height. After changing the window size, this method sends an `AdjustToEnclosure` message to its panes. This method respects the maximum and minimum window sizes you set with `SetSizeRect`.

MoveOffScreen

```
procedure MoveOffScreen;
void MoveOffScreen (void);
```

Move the window out of the visible area of the desktop.

Drawing methods

Update

```
procedure Update;
void Update (void);
```

Update the window. This method is invoked when the Switchboard processes an update event. This method sends a `Prepare` message to the window and then sends a `Draw` message to each of the window's panes. You should not override this method.

Prepare

```
procedure Prepare;
void Prepare (void);
```

Set the port of the window and prepare for drawing.

Mouse methods

DispatchClick

```
procedure DispatchClick (var macEvent: EventRecord);
void DispatchClick (EventRecord *macEvent);
```

Find a pane to handle the mouse click. You should not use or override this method. If you want to handle a click in a particular window subclass, override the `DoClick` method (inherited from CView) instead.

DispatchCursor

```
procedure DispatchCursor (where: Point;
   mouseRgn: RgnHandle);
void DispatchCursor (Point where, RgnHandle mouseRgn);
```

Find a pane that handles `AdjustCursor` messages. This method converts `where` from global to window coordinates, and then lets the `DispatchCursor` method in CView take care of displaying Balloon Help it if is enabled. You should not use or override this method.

Conversion methods

FrameToGlobalR

```
procedure FrameToGlobalR (frameRect: LongRect;
   globalRect: Rect);
void FrameToGlobalR (LongRect *frameRect,
   Rect *globalRect);
```

Convert `frameRect` from frame to global coordinates. Place the result in `globalRect`.

TCL Library Routines 71

Introduction

The THINK Class Libary uses some library routines to deal with the Macintosh Tolbox, long coordinates, and memory allocation. The Starter seed projects are set up so you can use these utility routines.

For a detailed description of the exception handling mechanism, see Chapter 8, "Exception Handling."

For more information about memory management, see "Handling low memory situations" on page 164.

Toolbox Utilities

These routines make working with parts of the Macintosh Toolbox easier.

THINK C programmers

These functions are defined in `TBUtilities.c`. Be sure to include `TBUtilities.h` before you use these functions.

THINK Pascal programmers

These routines are defined in `TCL.p`.

QuickDraw Utilities

DrawSICN

```
procedure DrawSICN (SICNid, index: integer;
   location: Point);
void DrawSICN (short SICNid, short index,
   Point location);
```

Draw a small icon at the point `location`. `SICNid` is the resource id of a `SICN` resource. `Index` is the number of the small icon within the resource.

PinInRect

```
procedure PinInRect (theRect: LongRect;
   var thePoint: LongPt);
void PinInRect (LongRect *theRect, Point *thePoint);
```

Pin `thePoint` within `theRect`. This routine is similar to the Toolbox routine `PinRect` except that the point is changed in place, and this routine does not subtract 1 at the right and bottom edges.

Window Manager Utilities

BringBehind

```
procedure BringBehind (macWindow,
   behindWindow: WindowPtr);
void BringBehind (WindowPtr macWindow,
   WindowPtr behindWindow);
```

`BringBehind` moves `macWindow` so it is behind `behindWindow`. The CFWDesktop class uses this routine to place the frontmost application window behind all the floating windows.

IsDialogWindow

```
function IsDialogWindow (macWindow: WindowPeek):
   Boolean;
Boolean IsDialogWindow (WindowPeek macWindow);
```

Return `TRUE` if `macWindow` is a dialog window.

IsMyWindow

```
function IsMyWindow (macWindow: WindowPeek):
   Boolean);
Boolean IsMyWindow (WindowPeek macWindow);
```

Returns `TRUE` if `macWindow` is an application window or a dialog window.

IsMyWindow

```
function IsSystemWindow (macWindow: WindowPeek):
   Boolean);
Boolean IsSystemWindow (WindowPeek macWindow);
```

Returns `TRUE` if `macWindow` is a system window (desk accessory window).

Dialog Manager Utilities

FindDlogPosition

```
procedure FindDlogPosition (theType: ResType;
   theID: integer; var corner: Point);
void FindDlogPosition (ResType theType, short theID,
    Point *corner);
```

Return in `corner` the coordinates of the upper left corner of the dialog or an alert if the dialog were centered in the upper third of the screen. This routine is useful for the standard file dialogs which ask you to supply a corner point. Otherwise, you should use `PositionDialog`.

PosistionDialog

```
procedure PositionDialog (theType: ResType;
   theID: integer);
void PositionDialog (ResType theType, short theID);
```

Center the bounding box of a dialog or an alert in the upper third of the screen. `TheType` is either `'DLOG'` or `'ALRT'` and `theID` is the resource ID of the dialog or the alert. `PositionDialog` does not display the dialog, it just changes its location.

Font Manager Utility

GetFontNumber

```
procedure GetFontNumber (fontName: ConstStr255Param;
   var fontNum: integer);
void GetFontNumber (ConstStr255Param fontName,
   short *fontNum);
```

Given a font name, return its font number in `fontNum` If the font is not found, return a negative number.

Keyboard Utilities

KeyIsDown

```
function KeyIsDown (theKeyCode: integer): Boolean;
Boolean KeyIsDown (short theKeyCode);
```

Return `TRUE` if the specifed key is being held down. Note that the key code is depends on the kind of keyboard. This function does not tell you if a specific key character is being held down.

AbortInQueue

```
function AbortInQueue: Boolean;
Boolean AbortInQueue (void);
```

Return `TRUE` if there is a Command-Period pending in the event queue. If there is, the event is removed from the queue.

IsCancelEvent

```
function IsCancelEvent (anEvent: EventRecord):
   Boolean;
Boolean IsCancelEvent (EventRecord *anEvent);
```

Return `TRUE` if anEvent is a "cancel" event. In US keyboards, a cancel event is Command-Period. For more information about cancelling in international environments, see Tech Note 263.

String Utilities

ConcatPStrings

C only

```
void ConcatPString (Str255 first,
   ConstStr255Param second);
```

Concatenate two Pascal strings. The string `second` is added to the end of `first`. The result is truncated if it would be longer than 255 bytes. Pascal programmers should use the built-in procedure `concat`.

CopyPString

C only

```
void CopyPString (ConstStr255Param srcString,
   Str255 destString);
```

Copy `srcString` into `desString`. The original contents of `destString` are lost. Pascal programmers should assign one string to another.

Operating System Utilities

THINK C programmers

These functions are defined in `OSChecks.c`. Be sure to include `OSChecks.h` before you use these functions.

THINK Pascal programmers

These routines are defined in `TCL.p`.

TrapAvailable

```
function TrapAvailable (trapNum: integer): Boolean;
Boolean TrapAvailable (short trapNum);
```

Return `TRUE` if the specified trap is available.

WNEIsImplemented

```
function WNEIsImplemented: Boolean;
Boolean WNEIsImplemented (void);
```

Return `TRUE` if the `WaitNextEvent` is available. `WaitNextEvent` is availble in every system from System 6.0.

TempMemCallsAvailable

```
function TemMemCallsAvailable: Boolean;
Boolean TemMemCallsAvailable (void);
```

Return `TRUE` if the MultiFinder temporary memory calls are available.

ColorQDIsPresent

```
function ColorQDIsPresent: Boolean;
Boolean ColorQDIsPresent (void);
```

Return `TRUE` if Color QuickDraw is available.

Long Coordinate Utilities

These routines operate on long rectangles and long points. These routines do not map long points from frame coordinates to QuickDraw coordinates or vice versa. To map frame coordinates to QuickDraw coordinates, use the transformation methods in CPane. See "Coordinate transformation methods" on page 357.

THINK C programmers

These functions are defined in `LongCoordinates.c`. Be sure to include `LongCoordinates.h` before you use these functions.

THINK Pascal programmers

These functions are defined in `TCL.p`.

Long point utilities

QDToLongPt

```
procedure QDToLongPt (srcPt: Point;
var destPt: LongPt);
void QDToLongPt (Point srcPt, LongPt *destPt);
```

Convert `srcPt` from a QuickDraw point to a long point and place the result in `destPt`.

LongToQDPt

```
procedure LongToQDPt (srcPt: LongPt;
   var destPt: Point);
void LongToQDPt (LongPt *srcPt, Point *destPt);
```

Convert `srcPt` from a long point to a QuickDraw point and place the result in `destPt`. If `srcPt` is not in QuickDraw space, it is clipped to 16 bits. To convert a long point in frame coordinates to a QuickDraw point, use CPane's `FrameToQD` method.

SetLongPt

```
procedure SetLongPt (var pt: LongPt; h, v: longint);
void SetLongPt (LongPt *pt, long h, long v);
```

Set the horizontal and vertical elements of long point `pt` to `h` and `v`.

AddLongPt

```
procedure AddLongPt (srcPt: LongPt;
   var destPt: LongPt);
void AddLongPt (LongPt *srcPt, LongPt *destPt);
```

Add `srcPt` and `destPt` and return the result in `destPt`.

SubLongPt

```
procedure SubLongPt (srcPt: LongPt;
   var destPt: LongPt);
void SubLongPt (LongPt *srcPt, LongPt *destPt);
```

Subtract `srcPt` from `destPt` and return the result in `destPt`.

EqualLongPt

```
function EqualLongPt (pt1: LongPt;
   pt2: LongPt): Boolean;
Boolean EqualLongPt (LongPt *pt1, LongPt *pt2);
```

Return `TRUE` if `pt1` and `pt2` are the same point.

PtInQDSpace

```
function PtInQDSpace (pt: LongPt): Boolean;
Boolean PtInQDSpace (LongPt *pt);
```

Return TRUE if pt is within 16-bit QuickDraw coordinate space.

Long rectangle utilities

QDToLongRect

```
procedure QDToLongRect (srcRect: Rect;
   var destRect: LongRect);
void QDToLongRect (Rect *srcRect, LongRect *destRect);
```

Convert `srcRect` from a QuickDraw rectangle to a long rectangle and return the result in `destRect`.

LongToQDRect

```
procedure LongToQDRect (srcRect: LongRect;
   var destRect: Rect);
void LongToQDRect (LongRect *srcRect, Rect *destRect);
```

Convert `srcRect` from a long rectangle to a QuickDraw rectangle and return the result in `destRect`.

SetLongRect

```
procedure SetLongRect (var r: LongRect; left, top,
right, bottom: longint);
void SetLongRect (LongRect *r, long left, long top,
long right, long bottom);
```

Set the elements of the long rectangle `r`.

OffsetLongRect

```
procedure OffsetLongRect (var r: LongRect; dh, dv:
longint);
void OffsetLongRect (LongRect *r, long dh, long dv);
```

Offset `r` by dh and dv. Positive values are to the right and down.

InsetLongRect

```
procedure InsetLongRect (var r: LongRect;
   dh, dv: longint);
void InsetLongRect (LongRect *r, long dh, long dv);
```

Inset the sides of `r` by dh and dv. Positive values move the sides in.

SectLongRect

```
function SectLongRect (src1, src2: LongRect;
   var destRect: LongRect): Boolean;
Boolean SectLongRect (LongRect *src1, LongRect *src2,
   LongRect *destRect);
```

Calculate the intersection of `src1` and `src2`, and place the result in `destRect`. `DestRect` may be the same as `src1` or `src2`. If `destRect` is not empy, return TRUE.

UnionLongRect

```
procedure UnionLongRect (src1, src2: LongRect;
   var destRect: LongRect);
void UnionLongRect (LongRect *src1, LongRect *src2,
   LongRect *destRect);
```

Calculate the union of `src1` and `src2`, and place the result in `destRect`. `DestRect` may be the same as `src1` or `src2`.

PtInLongRect

```
function PtInLongRect (pt: LongPt;
   r: LongRect): Boolean;
Boolean PtInLongRect (LongPt *pt, LongRect *r);
```

Return TRUE if the long point pt is in the long rectangle `r`.

Pt2LongRect

```
procedure Pt2LongRect (pt1, pt2: LongPt;
   var r: LongRect);
void UnionLongRect (LongPt *pt1, LongPt *pt2,
   LongRect *r);
```

Calculate the smallest rectangle that encloses `pt1` and `pt2` and return the result in the long rectangle `r`.

EqualLongRect

```
function EqualLongRect (r1, r2: LongRect): Boolean;
Boolean EqualLongRect (LongRect *r1, LongRect *r1);
```

Return TRUE if the long rectangles `r1` and `r2` are identical.

EmptyLongRect

```
function EmptyLongRect (r: LongRect): Boolean;
Boolean EmptyLongRect (LongRect *r);
```

Return TRUE if the long rectangle `r` encloses no points.

RectInQDSpace

```
function RectInQDSpace (r: LongRect): Boolean;
Boolean RectInQDSpace (LongRect *r);
```

Return TRUE if the long rectangle `r` is .entirely within QuickDraw space.

THINK Class Library Utilities

These routines are generally useful for working with the THINK Class Library. Some of them work with the exception handling mechanism (see Chapter 8, "Exception Handling") and with the memory management routines (see "Handling low memory situations" on page 164).

THINK C programmers

These functions are defined in `TCLUtilities.c`. Be sure to include `TCLUtilities.h` before you use these functions.

THINK Pascal programmers

These routines are defined in TCL.p. EqualMem is written in assembly language. It is defined in TCL.lib.

Error reporting utility

ErrorAlert

```
procedure ErrorAlert (error: integer;
   message: longint);
void  ErrorAlert (short error, long message);
```

Display an alert for the given error and message codes. If the low word of message is zero, ErrorAlert looks for an 'Estr' resource that matches the error code in error. If it can't find one, it displays a generic error message and the error number.

If the low word of message is greater than zero, it is used as an index into a 'STR#' resource. If the high word of message is zero, ErrorAlert uses the resource 'STR#' 301. If the high word of message is not zero, its value is added to 1024 to get the resource ID of a 'STR#' resource. You can use the exception handler's SpecifyMsg function to build the message.

For instance, if message is 655,363—0x000A0003 hex—, the resource ID of the 'STR#' resource is 1024+10, and the index into it is 3.

If the application is running, ErrorAlert uses 'ALRT' ALRT_Exception (251), otherwise it uses 'ALRT' ALRT_ExceptionAbort (252).

Memory allocation utilities

NewHandleCanFail

```
function NewHandleCanFail (size: longint): Handle;
void  *NewHandleCanFail (long size);
```

Allocate a handle without drawing on the memory reserve. If the allocation fails, returns NIL.

ResizeHandleCanFail

```
procedure ResizeHandleCanFail (theHandle: Handle;
   newSize: longint);
void ResizeHandleCanFail (void *theHandle, long
newSize);
```

Resize a handle without drawing on the memory reserve.

SetAllocation

```
function SetAllocation (canFail: Boolean): Boolean;
Boolean SetAllocation( Boolean canFail);
```

Change the parameters that gApplication uses when the grow zone function is invoked because the memory manager can't satisfy a request. If canFail is TRUE (or kAllocCanFail), memory requests are allowed to

fail without drawing on memory reserves. If canFail is FALSE (or kAllocCantFail), the application will try to satisfy the memory request from the reserves. SetAllocation returns the previous setting. You should reset the allocation parameters after you make a memory request.

In THINK C, you might do it like this:

```
void AClass::AMethod(void)
{
   Boolean oldAlloc;

   oldAlloc = SetAllocation(kAllocCanFail);
      request memory here
   SetAllocation(oldAlloc);
      fail gracefully if it doesn't succeed
}
```

In THINK Pascal, you could do it this way:

```
procedure AClass.AMethod;
var
   oldAlloc: Boolean;
begin
   oldAlloc := SetAllocation(kAllocCanFail);
      request memory here
   oldAlloc := SetAllocation(oldAlloc);
      fail gracefully if it doesn't succeed
end;
```

This method sends a RequestMemory method to gApplication. See "Handling low memory situations" on page 156 for more information about working with memory in the THINK Class Library.

SetCriticalOperation

```
procedure SetCriticalOperation (aCriticalOp: Boolean);
void SetCriticalOperation (Boolean aCriticalOp);
```

Change the parameters that gApplication uses when the grow zone function is invoked because the memory manager can't satisfy a request. If aCriticalOp is TRUE, more of the memory reserve is available to satisfy a failing memory request.

This method sends a SetCriticalOperation message to gApplication. See "Handling low memory situations" on page 156 for more information about working with memory in the THINK Class Library.

EqualMem

```
function EqualMem (p1, p2: Ptr; n: longint): Boolean;
Boolean EqualMem (void *p1, void *p2, long n);
```

Compare two blocks of memory, and return TRUE if they are equal. `P1` and `p2` point to memory, and `n` is the number of bytes to compare.

SetMinimumStack

```
procedure SetMinimumStack (newSize: minSize);
void SetMinimumStack (long minSize);
```

Set the stack to be at least `minSize` bytes. If you use this routine, it must appear only once, and it must be the first statement in your program.

Memory disposal utilities

These memory utilities release different kinds of memory, and set the variable that references them to NIL. In THINK C, these routines are implemented as macros, which are defined in `Global.h`. In THINK Pascal they're defined as procedures in `TCL.p`.

ForgetHandle

```
procedure ForgetHandle (var handle: Handle);
void ForgetHandle (Handle h); [MACRO]
```

If `h` is not NIL, call `DisposHandle(h)`, and set `h` to NIL.

ForgetPtr

```
procedure ForgetPtr (var p: Ptr);
void ForgetPtr (Ptr p); [MACRO]
```

If `p` is not NIL, call `DisposPtr(p)`, and set `p` to NIL.

ForgetResource

```
procedure ForgetResource (var r: Handle);
void ForgetResource (Handle r); [MACRO]
```

If `r` is not NIL, call `ReleaseResource(r)`, and set `r` to NIL.

ForgetObject

```
procedure ForgetObject (var obj: CObject);
void ForgetObject (CObject obj); [MACRO]
```

If `obj` is not NIL, send `obj` a `Dispose` or `Free` message, and set `obj` to NIL.

Long Coordinate QuickDraw Utilities

The following routines are long coordinate versions of some common QuickDraw routines. The arguments that these routines take are long points and long rectangles in frame coordinates. If you use these routines, be sure that you call them only in a pane's `Draw` method because they expect that the pane has already been prepared.

Note

These routines transform frame coordinates to QuickDraw coordinates every time you call them. It's usually more convenient to do the transformation once, then do all the drawing in QuickDraw coordinates.

THINK C programmers

These functions are defined in `LongQD.c`. Be sure to include `LongQD.h` before you use these functions.

THINK Pascal programmers

These routines are defined in `LongQD.p`.

Long rectangle drawing routines

LEraseRect

```
procedure LEraseRect (area: LongRect);
void LEraseRect (LongRect *area);
```

Erase the rectangle specified in `area`.

LFrameRect

```
procedure LFrameRect (area: LongRect);
void LFrameRect (LongRect *area);
```

Frames the rectangle specified in `area`.

LInvertRect

```
procedure LInvertRect (area: LongRect);
void LInvertRect (LongRect *area);
```

Invert the rectangle specified in `area`.

LPaintRect

```
procedure LPaintRect (area: LongRect);
void LPaintRect (LongRect *area);
```

Paint `area` in the current pattern and mode.

LFillRect

```
procedure LFillRect (area: LongRect; pat: Pattern);
void LFillRect (LongRect *area, Pattern pat);
```

Fill `area` in the pattern `pat` in `patCopy` mode.

Long oval drawing routines

LEraseOval

```
procedure LEraseOval (area: LongRect);
void LEraseOval (LongRect *area);
```

Erase the oval specified in `area`.

LFrameOval

```
procedure LFrameOval (area: LongRect);
void LFrameOval (LongRect *area);
```

Erase the oval specified in area.

LInvertOval

```
procedure LInvertOval (area: LongRect);
void LInvertOval (LongRect *area);
```

Invert the oval specified in area.

LPaintOval

```
procedure LPaintOval (area: LongRect);
void LPaintOval (LongRect *area);
```

Paint area in the current pattern and mode.

LFillOval

```
procedure LFillOval (area: LongRect; pat: Pattern);
void LFillOval (LongRect *area, Pattern pat);
```

Fill area in the pattern pat in patCopy mode.

Long rounded rectangle drawing routines

LEraseRoundRect

```
procedure LEraseRoundRect (area: LongRect;
   ovWidth, ovHeight: integer);
void LEraseRoundRect (LongRect *area,
   short ovWidth, short ovHeight);
```

Erase the rounded rectangle specified in area.

LFrameRoundRect

```
procedure LFrameRoundRect (area: LongRect;
   ovWidth, ovHeight: integer);
void LFrameRoundRect (LongRect *area,
   short ovWidth, short ovHeight);
```

Erase the rounded rectangle specified in area.

LInvertRoundRect

```
procedure LInvertRoundRect (area: LongRect;
   ovWidth, ovHeight: integer);
void LInvertRoundRect (LongRect *area,
   short ovWidth, short ovHeight);
```

Invert the rounded rectangle specified in area.

LPaintRoundRect

```
procedure LPaintRoundRect (area: LongRect;
   ovWidth, ovHeight: integer);
void LPaintRoundRect (LongRect *area,
   short ovWidth, short ovHeight);
```

Paint area in the current pattern and mode.

LFillRoundRect

```
procedure LFillRoundRect (area: LongRect;
   ovWidth, ovHeight: integer; pat: Pattern);
void LFillRoundRect (LongRect *area,
   short ovWidth, short ovHeight, Pattern pat);
```

Fill area in the pattern pat in patCopy mode.

Long bit transfer routine

LCopyBits

```
procedure LCopyBits (srcBits, dstBits: BitMap;
   srcRect, dstRect: LongRect;
   mode: integer; mask: RgnHandle);
void LCopyBits (BitMap *srcBits, BitMap *dstBits,
   LongRect *srcRect, LongRect *dstRect,
   short mode, RgnHandle mask);
```

Transfer a bit image from srcBits to dstBits. For a full description of this routine, see the description of CopyBits in *Inside Macintosh I.*

Long point pen routines

LMoveTo

```
procedure LMoveTo (hLoc, vLoc: longint);
void LMoveTo (long hLoc, long vLoc);
```

Move the pen to the point hLoc, vLoc.

LLineTo

```
procedure LLineTo (hLoc, vLoc: longint);
void LLineTo (long hLoc, long vLoc);
```

Draw a line from the current pen location to hLoc, vLoc.

Long region utility routines

LRectRgn

```
procedure LRectRgn (rgn: RgnHandle; rect: LongRect);
void LRectRgn (RgnHandle rgn, LongRect *rect);
```

Create a region whose shape is specified by rect. Rgn must be a handle to an existing region.

LClipRect

```
procedure LClipRect (r: LongRect);
void LClipRect (LongRect *r);
```

Change the clipping region of the current grafPort to the long rectangle r.

Global Variables 72

Introduction

This chapter describes the global variables in the THINK Class Library.

Global Objects

These globals hold pointers to the only instances of some classes. Most of them are initialized in the application initialization routines. You should not need to reset any of these variables yourself, except for `gSleepTime`.

gApplication

```
gApplication: CApplication;
CApplication *gApplication;
```

The global application object. You should set this variable to an instance of your application class in your main program. See "Writing the main program" in CApplication on page 163 for an example.

gDesktop

```
gDesktop: CDesktop;
CDesktop *gDesktop;
```

The desktop. This variable holds the only instance of CDesktop (or CFWDesktop if you're using floating windows). It's initialized in the application method `MakeDesktop`. The desktop is the enclosure for all the windows in your application. It is the top of the visual hierarchy.

gBartender

```
gBartender: CBartender;
CBartender *gBartender;
```

The bartender. This variable holds the only instance of CBartender. It's initialized in the application method `SetUpMenus`. The bartender converts menu choices into command numbers and handles most menu operations.

gClipboard

```
gClipboard: CClipboard;
CClipbard *gClipboard;
```

The clipboard. This variable holds the only instance of CClipboard. It's initialized in the application method `MakeClipboard`. The clipboard is where data is cut and copied to and pasted from.

gGopher

```
gGopher: CBureaucrat;
Cbureaucrat *gGopher;
```

The gopher is the first bureaucrat to get commands. If the gopher can't handle the command, it passes control to its supervisor. Initially, the gopher is set to the application (`gApplication`). When a document becomes active, the gopher points to the document. A document can set the gopher to point to one of its panes.

gError

```
gError: CError;
CError *gError;
```

The global error handler. This variable is initialized in `IApplication`.

gDecorator

```
gDecorator: CDecorator;
CDecorator *gDecorator;
```

The window dresser. This variable holds the only instance of CDecorator. It's initialized in the application method `MakeDecorator`. The decorator takes care of arranging windows on the screen.

Mouse Click Globals

The THINK Class Library uses these globals to count mouse clicks. The only variable you'll need to use is `gClicks`.

gLastMouseDown

```
gLastMouseDown: EventRecord;
EventRecord gLastMouseDown;
```

Event record of the last mouse-down event.

gLastMouseUp

```
gLastMouseUp: EventRecord;
EventRecord gLastMouseUp;
```

Event record of the last mouse-up event.

gLastViewHit

```
gLastViewHit: CView;
CView *gLastViewHit;
```

The last view the mouse went down in.

gClicks

```
gClicks: integer;
short gClicks;
```

Click counter. This variables counts multiple clicks. Its value is 1 for a single click, 2 for a double click, 3 for a triple click, etc. To be considered a multiple click the mouse must go down in the same view as the last mouse down within the amount of time specified by `GetDblTime` and the view method `HitSamePart` must return `TRUE`.

Cursors

These cursor handles are initialized in `IApplication`. You can use them as arguments to `SetCursor`. Remember that these are handles, so you'll have to call `SetCursor` like this in THINK Pascal:

```
SetCursor(gWatchCursor^);
```

And like this in THINK C:

```
SetCursor(*gWatchCursor);
```

gIBeamCursor

```
gIBeamCursor: CursHandle;
CursHandle gIBeamCursor;
```

I-beam for text views.

gWatchCursor

```
gWatchCursor: CursHandle;
CursHandle gWatchCursor;
```

Watch cursor for waiting.

System Globals

You can use these globals to get information about the environment that your application is running in, and about the state of your application.

gSystem

```
gSystem: tSystem;
tSystem gSystem;
```

This global is a record that contains fields that tell you about the capabilities of Macintosh that your program is running under. It's initialized in the `InspectSystem` method of CApplication.

In THINK C, `tSystem` is declared like this:

```
typedef struct
{
   Boolean  hasWNE          : 1;
   Boolean  hasColorQD      : 1;
   Boolean  hasGestalt      : 1;
   Boolean  hasAppleEvents  : 1;
   Boolean  hasAliasMgr     : 1;
   Boolean  hasEditionMgr   : 1;
   Boolean  hasHelpMgr      : 1;
   Boolean  hasScriptMgr    : 1;
   Boolean  hasFPU          : 1;
   short    scriptsInstalled;
   short    systemVersion;
} tSystem;
```

In THINK Pascal it's defined like this:

```
type
   tSystem = packed record of
      hasWNE,
      hasColorQD,
      hasGestalt,
      hasAppleEvents,
      hasAliasMgr,
      hasEditionMgr,
      hasHelpMgr,
      hasScriptMgr,
      hasFPU            : Boolean;
      scriptsInstalled,
      systemVersion     : integer;
   end;
```

The field `scriptsInstalled` tells you how many scripts are in use. The field `systemVersion` gives you the version of the Macintosh System that's running. The version number is given as two byte-long numbers. For example, if `systemVersion` is `0x0607`, the System version number is 6.0.7.

gSleepTime

```
gSleepTime: longint;
long gSleepTime;
```

The switchboard uses this value to pass to `WaitNextEvent`. It is the maximum time (in ticks) between events. Perform methods in CChore subclasses and `Dawdle` methods in CBureaucrat subclasses can change this value indirectly to force an idle event.

gInBackground

```
gInBackground: Boolean;
Boolean gInBackground;
```

`TRUE` if the application is in the background.

Error Globals

These global variables let you see what error raised the last exception. The values of these variables is valid only within a catch handler. At any other time, they're undefined. For more information about exception handling, see Chapter 8, "Exception Handling."

gLastError

```
gLastError: integer;
short gSignature;
```

The error that raised an exception.

gLastMessage

```
gLastMesage: longint;
long gLastMessage;
```

The message passed to `Failure` at the last exception.

Utility Globals

gSignature

```
gSignature: OSType;
OSType gSignature;
```

The signature of your application. You should initialize this variable in your `SetUpFileParameters` application method. Use this variable whenever your application needs to create a file.

gUtilRgn

```
gUtilRgn: RgnHandle;
RgnHandle gUtilRgn;
```

Utility region. This region is initialized with `NewRgn` in `IApplication`. You can use it wherever you need a region, but you should not rely on it being the same across calls to different methods.

Warning

If you use `gUtilRgn`, be sure that you never dispose of it. If you've created a complex region, and you want to release some memory, use `SetEmptyRgn` to make it as small as possible.

Index

Entries in **bold face** refer to menu commands. Entries in `typewriter face` refer to functions, methods, variables, keywords, or files.

Symbols

: : (scope resolution operator) 36
__TCL__DEBUG__ 121
16-bit coordinates 105, 106–107
32-bit coordinates 105, 106–107

A

AbortInQueue 487
AboutToPrint
  CDocument 292
  CEditText 302
  CPane 355
  CPanorama 376
abstract classes 93
AbTx resource 113
access control 43–48
  adjusting 62
  class variables, and 39
  creating objects, and 50
  specifier for superclass 45
accessing
  class variables 39
  instance variables 35
  methods 35
Activate 98
  CControl 260
  CDesktop 270
  CDirector 278
  CEditText 298
  CFWDesktop 316
  CScrollBar 411
  CSizeBox 428
  CView 466
  CWindow 481
activate events 98
ActivateDirector
  CDirector 278
  CDirectorOwner 282
ActivateWind
  CDirector 279
Add
  CCluster 244
AddDependent
  CCollaborator 254
AddDirector
  CDirectorOwner 282
AddLongPt 489
AddMenu
  CBartender 199
AddProvider
  CCollaborator 254
AddSubview
  CView 471
AddWind
  CDesktop 273
  CFWDesktop 317
AdjustBounds
  CEditText 301
AdjustCursor 113
  CAbstractText 156
  CDesktop 272
  CFWDesktop 317
  CView 468
AdjustHoriz
  CPane 352
AdjustScrollMax
  CScrollPane 416
AdjustToEnclosure
  CPane 352
AdjustVert
  CPane 352

alert resources 122
allocators 52
  default 54
ALRT resources 122
ancestor, *See* superclass
Append
  CList 324
Apple menu 115, 124
AppleEventIdle
  CSwitchboard 434
applications
  commands, handling 101
  subclass, writing your 100
  writing 98–103
Art Class Demo folder 12
ASSERT macro 121
Assert procedure 122
AssignIdleChore
  CApplication 184
assignment, objects 33
AssignUrgentChore
  CApplication 184
auto key events 98
AutoScroll
  CPanorama 375

B

bartender 93, 97
base class, *See* superclass
BecomeGopher
  CAbstractText 150
  CBureaucrat 217
BeginDrawing
  CBitMap 208
BeginTracking
  CMouseTask 337
bounds rectangle 110
BringBehind 486
BringFront
  CList 324
BroadcastChange
  CBureaucrat 217
  CCollaborator 253
Browser 81–86
bureaucrats 96

C

C++ 27, 60–62
CAbstractText 143
  resources, reading from 113
CalcAperture
  CPane 357
CalcBorderRect
  CPaneBorder 366
CalcFrame
  CPane 356
CalcTERects
  CEditText 301
CalcTopFloat
  CFWDesktop 318
Calibrate
  CScrollPane 416
CanBeGopher
  CView 466
CancelDependency
  CCollaborator 253
CancelIdleChore
  CApplication 184
CancelTyping
  CTextEditTask 450
canFail 119
CanStillType
  CTextEditTask 449
CAppleEvent 157
CApplication 163
  See also applications
CArray 185
catch handler 129
  in THINK C 132
  in THINK Pascal 135
CATCH macro 132
CatchFailure 135, 136
CBarOwner 8
CBartender 191
CBitMap 205
CBitMapPane 209
CBorder 8
CBureaucrat 213
CButton 219
CCharGrid 223
CCheckBox 227
CChore 231
CClipboard 235
CCluster 8, 243
CCollaborator 249
  See also collaborators
CCollection 255
CColorWindow 8
CControl 257
CDataFile 8, 263
CDecorator 267
CDesktop 269
CDirector 275
  See also directors

CDirectorOwner 281
CDocument 285
  See also documents
CEditText 295
  resources, reading from 113
CenterWindow
  CDecorator 268
CenterWithinEnclosure
  CPane 353
CEnvironment 305
CError 307
CFile 8, 311
CFWDesktop 315
CGridSelector 319
chain of command 96
ChangeName
  CFile 314
ChangeSelection
  CSelector 421
ChangeSize
  CControl 260
  CPane 352
  CScrollPane 416
  CWindow 482
CheckAllocation 309
checking menu items 116
CheckInsertion
  CEditText 299
CheckMarkCmd
  CBartender 201
CheckOSError
  CError 308
CheckResource 309
ChooseFile
  CApplication 183
ChooseItem
  CMenuDefProc 333
  CSelectorMDEF 426
circular class references 41
Class Browser 81–86
class keyword 31
  access control, and 45
class methods 40
class variables 38
  naming conventions 92
class_name 60, 62
classes 20, 28, 30–32
  abstract 93
  access specifier for superclass 45
  circular references 41
  finding declaration 81–82
  forward references 41
  name of, finding 60
  naming conventions 92
  organizing 55
  specifying when accessing member 36
  testing object for membership 60
  which to define 23
"Classes are indirect by default" option 59
Cleanup
  CDesktop 274
  CFWDesktop 318
clicks 97
  drawing, and 107
  receiving in pane 103
CList 8, 323
Clone
  CObject 340
Close
  CClipboard 238
  CDataFile 265
  CDirector 279
  CDirectorOwner 283
  CDocument 290
  CFile 313
  CResFile 404
  CWindow 478
ClosePrintMgr
  CPrinter 392
CloseWind
  CClipboard 238
  CDirector 279
  CDocument 290
  CTearOffMenu 444
CMBarChore 327
cmdNull 114
CMenuDefProc 329
CMouseTask 335
CNTL resources 123
CObject 339
code resources, and objects 28
collaborators 97
ColorQDIsPresent 488
Command keys, handling 98, 114
command numbers 114
  menu resources, in 115
  negative 116
command, chain of 96
commands 97
  application, handling 101
  document, handling 101
  menu, handling 114
  See also events

Compatibility clases 8, 11
ConcatPStrings 487
ConfirmClose
CDocument 290
constants
naming conventions 92
constructors 48–50
C++, and 62
Contains
CDesktop 272
CPane 350
CView 465
CWindow 480
Control classes folder 11
control resources 123
control, flow of 97
conventions, naming 92
ConvertGlobal
CClipboard 241
converting coordinates 106–107
ConvertPrivate
CClipboard 241
coordinate systems 105
panes, and 106–107
panoramas, and 110
Copy
CObject 340
CopyFrom
CBitMap 208
CopyFromTemporary
CArray 190
CopyPString 488
CopyTextRange
CAbstractText 152
CopyTo
CBitMap 208
CopyToTemporary
CArray 190
core classes 93
Core classes folder 11
CPane 341
See also panes
CPaneBorder 361
CPaneMDEF 367
CPanorama 371
See also panoramas
CPatternGrid 377
CPictFile 381
CPicture 383
resources, reading from 113
CPNTGFile 387
CPrinter 389
CRadioButton 8
CRadioControl 397
CRadioGroup 8
CRadioGroupPane 399
CreateDocument 100
CApplication 183
CreateNew
CFile 313
CResFile 404
creating
objects 32, 48–50, 60
CResFile 403
criticalBalance 118
CRunArray 405
CScrollBar 409
CScrollPane 413
See also scroll panes
CSelector 419
CSelectorMDEF 425
CSizeBox 427
CStack 429
CStaticText 8
CSupervisor
See also supervisors
CSwitchboard 431
CTask 437
See also tasks
CTearChore 441
CTearOffMenu 443
CTextEditTask 447
CTextEnvirons 453
CTextStyleTask 457
cursor tracking 113
See also mouse tracking
CView 461
See also views
CWindow 475

D

data member, *See* instance variable
DataSize
CClipboard 240
Dawdle
CBureaucrat 216
CEditText 302
Deactivate 98
CControl 260
CDesktop 270
CDirector 278
CEditText 298
CFWDesktop 316
CScrollBar 411
CSizeBox 428
CView 467
CWindow 481
DeactivateDirector
CDirector 278
CDirectorOwner 282
DeactivateWind
CDirector 279
deallocators 53
default 54
DebugExceptions 122
debugging 28, 121–122
THINK C aids 121
THINK Pascal aids 122
declared type 33
declaring
class variables 38
classes 30–32
instance variables 37
methods 39
objects 32
root class 30
subclass 31
See also defining
defining
class variables 38
methods 42
See also declaring
delete operator 32
calling method when deleting object 50–51
constructors, and 50
memory allocation, and 51–54
DeleteAll
CRunArray 407
DeleteFromBar
CBartender 199
DeleteItem
CArray 188
DeleteRange
CTextEditTask 450
DeleteRun
CRunArray 408
DeleteValue
CRunArray 407
deleting
objects 32, 50–51
dependents 97
DependUpon
CCollaborator 253
dereferencing objects 34
derived class, *See* subclass
desk accessories,
and objects 28
and THINK Class Library 98
desktop 95
destructors 50–51
C++, and 62
Dialog classes folder 11
dialog resources 122
dimALL 117
dimming menu items 116
dimNONE 117
dimSOME 117
direct classes 31, 34, 58
allocators, and 52
C++, and 62
direct commands 93
directive, #pragma once 55
directors 97
DisableCmd
CBartender 200
DisableMenu
CBartender 200
DisableMenuBar
CBartender 200
Dispatch
CSwitchboard 435
DispatchClick 97
CDesktop 271
CView 467
CWindow 483
DispatchCursor
CDesktop 272
CView 468
CWindow 483
dispatcher, message 41, 43
Dispose
CAbstractText 146
CArray 187
CBitMap 207
CBitMapPane 211
CBureaucrat 214, 384
CCharGrid 226
CCluster 244
CCollaborator 252
CControl 258
CDesktop 270
CDirector 277
CDirectorOwner 282
CDocument 288
CEditText 297
CFile 312
CFWDesktop 316

CObject 340
CPane 349
CPatternGrid 380
CPictFile 382
CPNTGFile 388
CPrinter 392
CSizeBox 428
CTextEditTask 449
CView 464
CWindow 477
in document 101
in pane 103
DisposeAll
CCluster 244
DisposeItems
CCluster 244
DisposHandle, and objects 54
DisposPtr, and objects 54
DITL resources 122
Do
CTask 439
CTextEditTask 449
CTextStyleTask 458
DoActivate
CSwitchboard 433
DoAppleEvent
CApplication 177
CBureaucrat 216
DoAutoKey 98
CAbstractText 149
CApplication 174
CBureaucrat 215
DoBackspace
CTextEditTask 450
DoClick 97
CControl 261
CEditText 297
CScrollBar 411
CSelector 421
CView 467
pane, for your 103, 107
DoCommand
application, for your 101
CAbstractText 148
CApplication 175
CBureaucrat 215
CDirector 277
CDocument 289
document, for your 101
menus, for your 114, 116
documents 100, 101–102
commands, handling 101
subclass, writing your 101
See also files
DoDeactivate
CSwitchboard 433
DoDiskEvent
CSwitchboard 432
DoDoubleClick
CSelector 423
DoForEach
CCluster 246
DoForEach1
CCluster 247
DoFwdDelete
CTextEditTask 450
DoGoodClick
CButton 221
CCheckBox 229
CControl 262
CRadioControl 398
DoHighLevelEvent
CSwitchboard 434
DoHorizScroll
CScrollPane 416
DoIdle
CSwitchboard 433
DoKeyDown 98
CAbstractText 149
CApplication 174
CBureaucrat 215
CPanorama 376
DoKeyEvent
CSwitchboard 432
DoKeyUp
CApplication 175
CBureaucrat 215
DoMouseDown
CSwitchboard 432
DoMouseUp
CDesktop 272
CSwitchboard 432
CView 467
DonePrinting
CDocument 292
CEditText 302
CPane 355
CPanorama 376
DoNormalChar
CTextEditTask 450
DoOtherEvent
CSwitchboard 433
DoPageSetup
CPrinter 394

DoPrint
CPrinter 394
DoResume
CSwitchboard 433
DoRevert 102
CDocument 292
DoSave 102
CDocument 292
DoSaveAs
CDocument 292
DoSaveFileAs
CDocument 293
DoScroll
CScrollPane 417
DoSuspend
CSwitchboard 433
DoTask
CTask 439
CTextEditTask 449
CTextStyleTask 458
DoThumbDrag
CScrollPane 417
DoThumbDragged
CControl 261
CScrollBar 411
DoTyping
CTextEditTask 449
DoUpdate
CSwitchboard 433
DoVertScroll
CScrollPane 417
Down Arrow key 85
Drag
CWindow 481
DragWind
CDesktop 273
CFWDesktop 318
Draw 106
CBitMapPane 211
CControl 260
CEditText 297
CGridSelector 320
CPane 354
CPicture 385
CScrollBar 411
CSizeBox 428
DrawAll
CControl 260
CPane 354
DrawBorder
CPaneBorder 365
DrawGrid
CGridSelector 321
drawing, in pane 106
DrawItem
CCharGrid 226
CGridSelector 321
CPatternGrid 380
DrawMenu
CMenuDefProc 333
CPaneMDEF 368
DrawSICN 485
drivers, and objects 28

E

Edit menu 115, 124
EmptyGlobalScrap
CClipboard 241
EmptyLongRect 491
EmptyScrap
CClipboard 242
EnableCmd
CBartender 200
EnableMenu
CBartender 200
EnableMenuBar
CBartender 200
encapsulation 19
enclosures 95, 103
EnclosureScrolled
CPane 353
EnclToFrame
CPane 358
EnclToFrameR
CPane 358
EndDrawing
CBitMap 208
EndTracking 120
CMouseTask 337
ENDTRY macro 132
Enter key 85
EqualLongPt 489
EqualLongRect 491
EqualMem 494
error handling, *See* exception handling
error messages
displaying your own 132, 141
exception handling, and 131
strings, for THINK Class Library 123
error parameter 136
ErrorAlert 492
Estr resources 123

events
passing to objects 97
See also commands
exception handling 129–142
error messages, displaying 131, 141
Exit, and 139
memory allocation 131
objects, creating 131, 133, 137
propagating errors 135, 139
retrying try handler 135, 139
returning values 134
THINK C 132–135
THINK Pascal 135–139
Exceptions.h 132
ExistsOnDisk
CFile 313
Exit
CApplication 182
Exit, and exception handling 139
ExitApp
CApplication 182
ExtractFromDescList
CAppleEvent 160

F

FailInfo 136
FailMemError 141
FailNIL 131, 141
FailNILRes 141
FailOSErr 141
FailResError 141
Failure 140
Far CODE, and objects 28
Far DATA, and objects 28
fi variable 136
File classes folder 11
File menu 115, 124
files
classes in 55
documents, and 97
See also documents
FindCmdNumber 114, 116
CBartender 201
FindDlogPosition 486
FindIndex
CList 325
FindItem
CCluster 245
CGridSelector 321
CSelector 422
FindItem1
CCluster 245
FindItemBox
CGridSelector 322
FindItemText
CBartender 202
FindLine
CAbstractText 156
CEditText 302
FindMenuIndex
CBartender 202
FindMenuItem
CBartender 202
FindRun
CRunArray 408
FindSubview
CView 472
FindSum
CRunArray 407
FindViewByID
CDirector 277
CView 472
FirstItem
CList 325
FirstSuccess
CList 325
FirstSuccess1
CList 325
FitToEnclFrame
CPane 353
FitToEnclosure
CPane 353
flags
naming conventions 92
flow of control 97
Font menu 115, 124
ForceClassReferences
CApplication 173
ForceNextPrepare
CView 473
ForgetHandle 494
ForgetObject 494
ForgetPtr 494
ForgetResource 494
frame coordinates 105
frames 106
FrameToBounds
CPicture 386
FrameToEncl
CPane 358
FrameToEnclR
CPane 358

FrameToGlobalR
CPane 359
CView 473
CWindow 483
FrameToQD 106
CPane 359
FrameToQDR 106
CPane 359
FrameToWind
CPane 358
FrameToWindR
CPane 358
Free
CAbstractText 146
CArray 187
CBitMap 207
CBitMapPane 211
CBureaucrat 214, 384
CCharGrid 226
CCluster 244
CCollaborator 252
CControl 258
CDesktop 270
CDirector 277
CDirectorOwner 282
CDocument 288
CEditText 297
CFile 312
CFWDesktop 316
CObject 340
CPane 349
CPatternGrid 380
CPictFile 382
CPNTGFile 388
CPrinter 392
CSizeBox 428
CTextEditTask 449
CView 464
CWindow 477
in document 101
in pane 103
friends 62
function member, *See* method
FW/Tearoffs folder 11

G

gApplication 499
gAskFailure 121
gBartender 499
gBreakFailure 121
gClicks 501
gClipboard 500
gDecorator 500
gDesktop 499
GenericMDEF 334
gError 500
GetlHeight
CAbstractText 154
GetAEEvent
CAppleEvent 159
GetAERefCon
CAppleEvent 160
GetAEReply
CAppleEvent 160
GetAlignCmd
CAbstractText 154
CEditText 300
GetAnEvent
CSwitchboard 435
GetAperture
CDesktop 274
CPane 350
CView 465
CWindow 479
GetBalloonInfo
CView 471
GetBitMap
CBitMapPane 211
GetBorder
CPane 351
GetBorderFlags
CPaneBorder 364
GetBounds
CBitMap 207
CDesktop 274
CPanorama 374
GetBoundsOrigin
CBitMap 207
GetCharAfter
CAbstractText 152
GetCharBefore
CAbstractText 152
GetCharOffset
CAbstractText 154
CEditText 300
GetCharPoint
CAbstractText 155
CEditText 301
GetCharStyle
CAbstractText 155
CEditText 301
GetClassName
CObject 340

GetClickCmd
CButton 221
GetCmdText
CBartender 201
GetCommandBase
CSelector 422
GetData
CClipboard 241
GetDescList
CAppleEvent 160
GetErrorResult
CAppleEvent 161
GetEventClass
CAppleEvent 159
GetEventID
CAppleEvent 159
GetExtent
CPanorama 373
GetFontNumber 487
GetFrame
CPane 349
CView 465
CWindow 478
GetFramePosition
CPanorama 373
GetFrameSpan
CPanorama 373
GetFSSpec
CFile 313
GetGlobalScrap
CClipboard 239
GetHeight
CAbstractText 154
CEditText 300
GetHelpResID
CPane 351
CView 471
CWindow 480
GetHomePosition
CPanorama 375
GetID
CView 466
GetInterior
CScrollPane 416
CView 465
CWindow 478
GetItem
CArray 188
GetLength
CAbstractText 156
CDataFile 264
CEditText 302
CPane 350
GetMacPicture
CPicture 385
GetMacPort
CView 465
GetMacWindow
CTearOffMenu 445
GetMargin
CPaneBorder 365
GetMargins
CTearOffMenu 445
GetMark
CDataFile 265
GetMaxValue
CControl 258
GetMinValue
CControl 258
GetName
CDocument 293
CFile 313
GetNameIndex
CTask 438
GetNumItems
CCollection 255
CRunArray 406
GetNumLines
CAbstractText 156
CEditText 302
GetOrigin
CPane 350
CView 465
GetPageArea
CPrinter 394
GetPageInfo
CPrinter 394
GetPageStart
CPrinter 394
GetPattern
CPaneBorder 364
CPatternGrid 380
GetPenSize
CPaneBorder 364
GetPhase
CApplication 174
GetPixelExtent
CPane 350
CPanorama 375
GetPosition
CPanorama 374
GetPrintRecord
CPrinter 394

GetRounding
CPaneBorder 365
GetScaled
CPicture 385
GetScales
CPanorama 374
GetSelection
CAbstractText 151
CEditText 298
CSelector 422
GetShadow
CPaneBorder 365
GetSpacingCmd
CAbstractText 154
CEditText 300
GetSpecification
CAbstractText 147
GetStationID
CRadioGroupPane 401
GetSteps
CScrollPane 416
GetStripCount
CPrinter 393
GetSupervisor
CBureaucrat 214
GetTEFontInfo
CEditText 300
GetTextHandle
CAbstractText 151
CEditText 298
GetTextInfo
CTextEnvirons 455
GetTextRange
CEditText 299
GetTextStyle
CAbstractText 155
CEditText 301
GetTitle
CControl 259
CWindow 479
GetTopWindow
CDesktop 274
GetValue
CControl 258
CRunArray 407
GetWantsClicks
CView 465
GetWCount
CDecorator 268
GetWholeLines
CAbstractText 155
GetWindow
CDirector 277
CPane 351
GetXferMode
CBitMap 207
gGopher 500
gIBeamCursor 501
gInBackground 503
gLastError 503
gLastMessage 503
gLastMouseDown 500
gLastMouseUp 500
gLastViewHit 500
global coordinates 105
global variables
naming conventions 92
gopher 96, 97
GotRequiredParams
CAppleEvent 160
grow icon 124
GrowMemory 118
CApplication 179
GrowZoneFunc 308
gSignature 503
gSleepTime 502
gSystem 501
gUtilRgn 503
gWatchCursor 501

H

HandleFailure 136
handles, objects as 34
HasResFork
CResFile 404
HavePagination
CPrinter 392
header files
classes in 55
HGetState, and objects 57
Hide
CControl 259
CDesktop 270
CFWDesktop 316
CPane 352
CView 466
CWindow 480
HideSuspend
CWindow 481
HideWind
CDesktop 273
CFWDesktop 317

HiliteItem
CGridSelector 321
CPatternGrid 380
CSelector 422
HitSamePart
CDesktop 273
CSelector 421
CView 467
HLock, and objects 57
horizontal sizing characteristics, of pane 108
HSetState, and objects 57
HUnlock, and objects 57

I

IAbstractText
CAbstractText 145
IAppleEvent
CAppleEvent 159
IApplication 100, 118
CApplication 168
IArray
CArray 187
IBartender
CBartender 199
IBitMap
CBitMap 207
IBitMapPane
CBitMapPane 210
IBureaucrat
CBureaucrat 214
IButton
CButton 220
ICharGrid
CCharGrid 225
ICheckBox
CCheckBox 228
IClipboard
CClipboard 237
ICluster
CCluster 244
ICollaborator
CCollaborator 252
ICollection
CCollection 255
icon, small, resource 124
IDataFile
CDataFile 264
IDecorator
CDecorator 268
IDesktop
CDesktop 270
IDirector
CDirector 276
IDirectorOwner
CDirectorOwner 282
Idle
CApplication 182
IDocument 101
CDocument 288
IEditText
CEditText 296
IFile
CFile 312
IFWDesktop
CFWDesktop 316
IGridSelector
CGridSelector 320
IList
CList 323
IMenuDefProc
CMenuDefProc 333
IMouseTask
CMouseTask 337
Includes
CCluster 245
inCriticalOperation 118
indirect classes 31, 34, 58
allocators, and 52
C++, and 62
"Classes are indirect by default" option 59
INewButton
CButton 220
INewCheckBox
CCheckBox 228
INewRadioControl
CRadioControl 398
INewWindow
CWindow 477
inheritance 31
instance variables, and 37
multiple 62
inherited keyword 43
C++, and 62
InitMemory
CApplication 169
InitToolbox
CApplication 169
InsertAfter
CList 324
InsertAt
CList 324

InsertAtIndex
CArray 188
InsertHierMenu
CBartender 200
InsertInBar
CBartender 199
InsertMenuCmd
CBartender 201
InsertRun
CRunArray 408
InsertTextHandle
CAbstractText 152
InsertTextPtr
CAbstractText 152
CEditText 299
InsertValue
CRunArray 406
InsetLongRect 490
InspectSystem
CApplication 172
InstallPanorama 112
CScrollPane 415
InstallPatches
CApplication 173
instance variables 20, 28, 37–39
access control, and 43–48
accessing 35
declaring 37
inheritance, and 37
name, same as local variable 42
name, same as other instance variable 37
naming conventions 92
specifying class for 36
Toolbox calls, and 55
instances 20
See also objects
instantiation, *See* creating
integer type 92
IPane 102
CPane 347
IPaneBorder
CPaneBorder 364
IPaneMDEF
CPaneMDEF 368
IPanorama
CPanorama 372
IPatternGrid
CPatternGrid 379
IPictFile
CPictFile 381
IPicture
CPicture 384
IPNTGFile
CPNTGFile 387
IPrinter
CPrinter 391
IRadioControl
CRadioControl 398
IRadioGroupPane
CRadioGroupPane 400
IResFile
CResFile 403
IResPaneBorder
CPaneBorder 364
IRunArray
CRunArray 406
IsActive
CDirector 280
CView 464
IsCancelEvent 487
IsChecked
CCheckBox 229
IsColor
CWindow 479
IScrollBar
CScrollBar 410
IScrollPane
CScrollPane 414
IsDialogWindow 486
ISelector
CSelector 420
ISelectorMDEF
CSelectorMDEF 425
IsEmpty
CCollection 256
IsFloating
CWindow 479
ISizeBox
CSizeBox 427
IsModal
CWindow 479
IsMyWindow 486
IsOpen
CResFile 404
IsSystemWindow 486
IStack
CStack 429
IsUndone
CTask 438
IsVisible
CView 464

ISwitchboard
CSwitchboard 432
ITask
CTask 438
ITearChore
CTearChore 441
ITearOffMenu
CTearOffMenu 444
item number 116
ItemOffset
CArray 190
ITextEditTask
CTextEditTask 448
CTextStyleTask 458
ITextEnvirons
CTextEnvirons 455
IView
CView 463
IViewRes 113
CAbstractText 146
CEditText 296
CPane 349
CPanorama 373
CPicture 384
CScrollPane 415
CView 464
IViewTemp
CPane 349
CPanorama 373
CPicture 384
CScrollPane 415
CView 464
IWindow
CWindow 476

J

JumpToEventLoop
CApplication 183

K

KeepTracking 120
CMouseTask 337
key down events 98
KeyIsDown 487
keywords, for object extensions 61
kSilentError 132

L

"Language Extensions" option 58, 61
LastItem
CList 325
LastSuccess
CList 326
LastSuccess1
CList 326
LClipRect 497
LCopyBits 497
Left Arrow key 85
LEraseOval 495
LEraseRect 495
LEraseRoundRect 496
LFillOval 496
LFillRect 495
LFillRoundRect 497
LFrameOval 496
LFrameRect 495
LFrameRoundRect 496
libraries, for objects 28
LInvertOval 496
LInvertRect 495
LInvertRoundRect 496
LLineTo 497
LMoveTo 497
local coordinates 105
local variables, and duplicate names 42
location, of pane 103
Lock
CObject 340
locking objects 57
long coordinates 105, 106–107
long int type 92
longint type 92
LongPt 106
LongRect 106
LongToQDPt 489
LongToQDRect 490
LPaintOval 496
LPaintRect 495
LPaintRoundRect 496
LRectRgn 497

M

MakeBartender
CApplication 172
MakeClipboard
CApplication 170
MakeClipView
CClipboard 242
MakeCurrent
CResFile 404
MakeDecorator
CApplication 171
MakeDesktop
CApplication 170
MakeEditTask
CAbstractText 150
MakeError
CApplication 170
MakeMacTE
CEditText 297
MakeMacWindow
CWindow 478
MakePrinter
CDocument 291
MakeStyleTask
CAbstractText 150
MakeSwitchboard
CApplication 170
MatchView
CView 472
MBAR resource 124
member 20
member function, *See* method
member 60, 62
members
See also instance variables, methods
memory, allocating
objects, and 51–54, 55–57
exception handling, and 131, 141
for your document 101
for your pane 103
MemoryReplenished
CApplication 180
MemoryShortage 118
CApplication 179
menu bar resource 124
menu ID 116
MENU resources 115, 123
MENUapple 115
MENUedit 115
MENUfile 115
MENUfont 115
menus 114–118
adding items 116
checking items 116
clicking in 116
creating in code 116
dimming items 116
resources, reading from 115
MenuSelect 117
MENUsize 115
message dispatcher 41, 43
message parameter 136
messages 19
See also methods
methods 20, 28, 39–41, 42–43
access control, and 43–48
callback routines, as 40
calling 36, 41
creating objects, calling method when 48–50
declaring 39
defining 42
deleting object, calling method when 50–51
finding defintion 83–85
monomorphic 41
name, same as other method 39
overriding 39
specifying class for 36
static 40
superclass, calling 43
virtual 39
"Methods are virtual by default" option 59
-%_MethTables- 121
MissingResources
CError 308
monomorphic methods 41
"Optimize monomorphic methods" option 59
More classes folder 11
MoreSlots
CArray 190
mouse clicks, *See* clicks
mouse tracking 120
See also cursor tracking
Move
CWindow 482
MoveDown
CList 324
MoveItemToIndex
CArray 189
MoveOffScreen
CWindow 482
MoveToCorner
CTearOffMenu 445
MoveToIndex
CList 325
MoveUp
CList 324
MultiFinder, running under 98
multiple inheritance 62

N

name
  class, finding 60
  creating object by
  duplicate, and instance variables 37
  duplicate, local variables 42
  duplicate, methods 39
naming conventions 92
new operator 32
  access control, and 50
  calling method when creating object 48–50
  constructors, and 48
  exception handling, and 131
  memory allocation, and 51–54
New, handling 100, 102
new_by_name 60, 62
NewClassDemo Folder 12
NewFile 102
  CDocument 290
NewHandle, and objects 54
NewHandleCanFail 492
NewPtr, and objects 54
NIL 92
NO_PROPAGATE macro 135
non-virtual methods 39
Notify 119, 120
  CApplication 174
  CBureaucrat 215
  CDocument 288
NthItem
  CList 325
NULL 92

O

Object Pascal 27
object reference 32
objects 19, 28, 32–34
  allocating memory 34
  assignment 33
  class name, finding 60
  creating 32, 48–50, 60
  creating and exception handling 131, 133, 137
  declaring 32
  deleting 32
  dereferencing 34
  locking in memory 57
  memory allocation, and 51–54
  testing for class membership 60
  Toolbox calls, and 55
OCluster 8
ODataFile 8
Offset
  CCluster 245
  CControl 260
  CPane 352
OffsetLongRect 490
OFile 8
OList 8
once-only headers 55
oops library 28
oops.h 60
oopsA4 library 28
oopsDebug library 28
oopsFar library 28
Open
  CDataFile 265
  CFile 313
  CResFile 404
Open..., handling 100, 102
OpenDocument 100
  CApplication 183
OpenFile 102
  CDocument 291
OpenPrintMgr
  CPrinter 392
"Optimize monomorphic methods" option 59
Options... dialog 58
organizing classes 55
origin 110
OutOfMemory
  CApplication 180
overriding methods 21, 39
OwnsWindow
  CDirector 277

P

PackageAppleEvent
  CApplication 176
PageCount
  CDocument 291
PageNumToStrips
  CPrinter 394
Paginate
  CAbstractText 151
  CDocument 291
  CPane 355
  CPanorama 376
Pane resource 113
panes 103–113
  clicks, receiving 103
  coordinates 105, 106–107
  drawing 106
  initializing 102
  location, setting 103, 107–109
  memory, allocating 103
  moving 107
  resizing 107
  resources, reading from 113
  scrolling 109–112
  subclass, writing your 102
  supervisor, and 103
  windows, and 104
  See also windows
Pano resource 113
panoramas 109–112
  resources, reading from 113
parameters
  naming conventions 92
pascal, methods declared as 40
PctP resource 113
Perform
  CChore 232
  CMBarChore 327
  CTearChore 441
PerformEditCommand
  CAbstractText 149
  CEditText 297
PickFileName
  CDocument 293
PinInRect 486
PixelIsBlack
  CBitMap 207
Place
  CPane 352
PlaceNewWindow
  CDecorator 268
PlacePopUp
  CMenuDefProc 333
pointers, objects as 34
polymorphism 22
Pop
  CStack 429
position 110
PositionDialog 487
PostAlert
  CError 308
#pragma once directive 55
Preload
  CApplication 181
Prepare 105
  CControl 260
  CDesktop 274
  CPane 356
  CView 473
  CWindow 482
PrepareToPrint
  CPane 356
Prepend
  CList 324
PrintPage
  CEditText 302
  CPane 355
  CPanorama 376
PrintPageOfDoc
  CDocument 292
PrintPageRange
  CPrinter 395
private access 43–46
PrivateChanged
  CClipboard 242
ProcessEvent
  CApplication 181
ProcessEvent
  CSwitchboard 434
protected access 43–48
ProviderChanged
  CBureaucrat 217
  CCollaborator 253
  CDirector 280
  CRadioGroupPane 401
providers 97
Pt2LongRect 491
PtInLongRect 491
PtInQDSpace 490
PtInTearRgn
  CPaneMDEF 370
public access 43–46
Push
  CStack 429
PushTryHandler 140
PutData
  CClipboard 240
PutGlobalScrap
  CClipboard 239

Q

QDToFrame
  CPane 359
QDToFrameR
  CPane 359
QDToLongPt 489

QDToLongRect 490
QuickDraw coordinates 105, 106–107
Quit
  CApplication 182
  CDirectorOwner 283
Quit, handling 101

R

rainy day fund 118
raising exceptions, *See* exception handling
ReadAll
  CDataFile 265
  CPictFile 382
ReadNewBitMap
  CPNTGFile 388
ReadSome
  CDataFile 266
ReallyVisible
  CDesktop 271
  CPane 350
  CView 464
RectInQDSpace 491
Redo
  CTask 439
  CTextEditTask 449
Refresh 106
  CPane 354
RefreshBorder
  CPane 355
RefreshLongRect
  CPane 355
RefreshRect
  CPane 354
Remove
  CCluster 245
RemoveDependent
  CCollaborator 254
RemoveDirector
  CDirectorOwner 282
RemoveMenu
  CBartender 199
RemoveMenuCmd
  CBartender 201
RemovePatches
  CApplication 173
RemoveProvider
  CCollaborator 254
RemoveSubview
  CView 472
RemoveWind
  CDesktop 273
  CFWDesktop 317
RequestInteraction
  CAppleEvent 160
RequestMemory
  CApplication 177
ResEdit 13
ResetPagination
  CPrinter 392
Resize
  CArray 190
  CWindow 482
ResizeFrame
  CAbstractText 156
  CEditText 301
  CPane 356
  CPanorama 375
  CPicture 385
ResizeHandleCanFail 492
ResolveFileAlias
  CFile 313
resource files
  menus, reading from 115
  THINK Class Library, for 98, 122–125
  views, reading from 113
Restore
  CEnvironment 305
  CTextEnvirons 455
RestoreEnvironment
  CPane 356
RestoreQuickDraw
  CPaneMDEF 370
RestoreRange
  CTextEditTask 451
RestoreStyle
  CTextStyleTask 459
Resume
  CApplication 181
  CClipboard 238
  CDirector 279
  CDirectorOwner 283
  CTearOffMenu 444
resume events 98
ResumeAfterError 122
Retrieve
  CArray 189
RETRY macro 135
RetryException 140
retrying try handler 135, 139
Return key 85
returning values and exception handling 134
Revert, handling 102
Right Arrow key 85
root class 21, 30
.rsrc file, *See* resource file
Run
  CApplication 180
run time type 33

S

Save, handling 102
SaveRange
  CTextEditTask 450
SaveStyle
  CTextStyleTask 459
SBarActionProc 417
SBarThumbFunc 417
scale 110
scope resolution operator (::) 36
  unary scope resolution 62
ScPn resource 113
ScrapConverted
  CClipboard 240
Scroll
  CEditText 298
  CPanorama 375
scroll panes 112
scrolling 109–112
ScrollTo
  CPanorama 375
ScrollToSelection
  CAbstractText 150
  CPanorama 375
Search
  CArray 189
SectAperture
  CPane 359
SectLongRect 490
segmentation 120
Select
  CWindow 481
SelectAll
  CAbstractText 151
SelectionChanged
  CAbstractText 150
  CTextEditTask 450
SelectWind
  CDesktop 273
  CFWDesktop 317
self object 42
•%_SelProcs• 121
SendBack
  CList 324
SetActClick
  CWindow 479
SetActionProc
  CControl 259
SetAlignCmd
  CAbstractText 153
  CEditText 300
SetAlignment
  CEditText 300
SetAllocation 119, 492
SetAllStripHeights
  CPrinter 393
SetAllStripWidths
  CPrinter 393
SetBitMap
  CBitMapPane 211
SetBlockSize
  CArray 187
SetBorder
  CPane 351
SetBorderFlags
  CPaneBorder 364
SetBounds
  CPanorama 373
SetCanBeGopher
  CView 466
SetClickCmd
  CButton 221
SetCmdText
  CBartender 201
SetCommandBase
  CSelector 422
SetCriticalOperation 119, 493
  CApplication 178
SetCursor 113
SetDefault
  CButton 221
SetDimOption 117
  CBartender 202
SetErrorResult
  CAppleEvent 161
SetFailInfo 142
SetFontName
  CAbstractText 153
SetFontNumber
  CAbstractText 153
  CEditText 299
SetFontSize
  CAbstractText 153
  CEditText 299

SetFontStyle
CAbstractText 153
CEditText 299
SetFrameOrigin
CPane 349
SetGridOn
CGridSelector 322
SetHelpResID
CWindow 480
SetHorizPageBreak
CPrinter 393
SetID
CView 466
SetItem
CArray 188
SetLength
CDataFile 264
SetLockChanges
CArray 187
SetLongPt 489
SetLongRect 490
SetMacPicture
CPicture 385
SetMargin
CPaneBorder 365
SetMargins
CTearOffMenu 445
SetMark
CDataFile 264
SetMaxValue
CControl 258
SetMinimumStack 494
SetMinValue
CControl 258
SetModal
CWindow 479
SetOverlaps
CScrollPane 416
SetPattern
CPaneBorder 364
SetPenSize
CPaneBorder 364
SetPosition
CPanorama 374
SetPrintClip
CPane 351
SetPrintDir
CPrinter 392
SetResBorder
CPane 351
SetRounding
CPaneBorder 365
SetScaled
CPicture 385
SetScales
CPanorama 374
SetScrollPane
CPanorama 374
SetSelection
CAbstractText 150
CEditText 298
SetShadow
CPaneBorder 365
SetSizeRect
CWindow 480
SetSpacingCmd
CAbstractText 154
CEditText 300
SetStationID
CRadioGroupPane 401
SetStdState
CWindow 480
SetSteps
CScrollPane 415
SetStripHeight
CPrinter 393
SetStrips
CPrinter 392
SetStripWidth
CPrinter 393
SetTextHandle
CAbstractText 151
SetTextInfo
CTextEnvirons 455
SetTextMode
CAbstractText 153
CEditText 299
SetTextPtr
CAbstractText 151
CEditText 298
SetTextString
CAbstractText 151
SetThumbFunc
CScrollBar 410
SetTitle
CControl 259
CWindow 479
SetUnchecking 117
CBartender 203
SetUpFileParameters 100
CApplication 171
SetUpMenus 115
CApplication 172

SetupQuickDraw
CPaneMDEF 370
SetValue
CControl 258
CRunArray 407
SetVertPageBreak
CPrinter 393
SetWantsClicks
CView 465
SetWholeLines
CAbstractText 155
SetXferMode
CBitMap 207
SevereMacError
CError 308
SFSpecify
CFile 312
short coordinates 105, 106–107
short int type 92
Show
CControl 260
CDesktop 270
CFWDesktop 316
CPane 351
CView 466
CWindow 480
ShowOrHide
CWindow 481
ShowResume
CWindow 481
ShowWind
CDesktop 273
CFWDesktop 317
SICN resource 124
SimulateClick
CButton 221
16-bit coordinates 105, 106–107
Size menu 115, 124
sizELASTIC 108
SizeMenu
CMenuDefProc 333
CPaneMDEF 369
sizeof 62
sizFIXEDBOTTOM 108
sizFIXEDLEFT 108
sizFIXEDRIGHT 108
sizFIXEDSTICKY 108
sizFIXEDTOP 108
sizing characteristics, of pane 107–109
small icon resource 124
source files
classes in 55
specifiers, access 43–48
Specify
CAbstractText 146
CFile 312
SpecifyFSSpec
CFile 312
SpecifyHFS
CFile 312
SpecifyMsg 141
StaggerWindow
CDecorator 268
Starter Folder 12
Starter project
building 12
writing program with 98–103
StartUpAction
CApplication 181
static
instance variables 38
methods 40
static data member, *See* class variable
static member function, *See* class method
Status
CClipboard 240
storage specifier 31
Store
CArray 188
StoreToClip
CTextEditTask 451
STR resources 124
STR# resources 125
string resources 124
strings, error messages 123
struct keyword 31
access control, and 45
"Classes are indirect by default" option, and 59
subclass 21, 31
access control, and 45
when to create 23
See also class
SubclassResponsibility
CObject 340
SubLongPt 489
SubpaneLocation
CView 472
Success 136, 139
SumRange
CRunArray 407
superclass 21, 31
access specifier for 45
calling method of 43

supervisors 96
panes, and 103
Suspend 98
CApplication 181
CClipboard 238
CDirector 278
CDirectorOwner 282
CTearOffMenu 444
suspend events 98
Swap
CArray 189
switchboard 93, 97
SwitchFromDA
CApplication 182
SwitchToDA
CApplication 182

T

Tab key 85
Table classes folder 11
tasks 119
TCL 1.1 Demos folder 12
TCL 1.1 Demos.sea 12
TCL 1.1 doc folder 12
__TCL__DEBUG__ 121
TCL Libraries folder 11
TCL Resources 98
TCL Resources folder 12
TCL TMPLs 113
installing 13
TCL TMPLs folder 12
TCL_DEBUG 122
TCLRuntime.lib
exception handling, and 131
TearOffMenu
CPaneMDEF 369
TempMemCallsAvailable 488
Text classes folder 11
"THINK C + Objects" option 58, 61
THINK Class Library 91–126
chain of command 96
class diagram 93
distributing 126
flow of control 97
modifying 125
naming conventions 92
resources 122–125
seed projects 12
visual hierarchy 95
THINK Class Library 1.1 folder 11
THINK Class Library 1.1.sea 11
32-bit coordinates 105, 106–107
this object 42
ThrowOut
CFile 314
TinyEdit Folder folder 12
TMPL resources 13
Toggle
CClipboard 238
Toolbox routines
exception handling, and 141
objects, and 55
pane coordinates for 105
toolboxBalance 118
TornOff
CTearOffMenu 445
tracking, cursor 113
tracking, mouse 120
TrackMouse
CView 357
TrapAvailable 488
try handler 129
retrying 135, 139
THINK C, in 132
THINK Pascal, in 135
TRY macro 132
Type
CEditText 297
type
C and Pascal 91
run time and declared 33
TypeChar
CAbstractText 149
type-compatible objects 33
typing, handling 98

U

unary scope resolution 62
Undo 119, 120
CTask 439
CTextEditTask 449
CTextStyleTask 458
undo 119
UnionLongRect 491
units, panorama 110
Up Arrow key 85
Update 98
CResFile 404
CWindow 482
update events 98, 106
UpdateAllMenus
CBartender 203
UpdateDisplay
CClipboard 239
UpdateMenuBar
CBartender 203
UpdateMenus 116
CAbstractText 148
CApplication 176
CBureaucrat 216
CDirector 278
CDocument 289
UpdateUndo
CDocument 293
UpdateWindows
CDesktop 273
UseLongCoordinates
CView 466
UsePICT
CPicture 385

V

variables
naming conventions 92
local, and duplicate names 42
vertical sizing characters, of pane 108
View resorce 113
views 95
resources, reading from 113
virtual methods 39
"Methods are virtual by default" option, and 59
visual hierarchy 95
visual messages 93

W

WantsActClick
CWindow 479
WIND resources 125
window coordinates 105
window resources 125
windows
documents, and 97
events, handling 97, 98
panes, and 104
See also panes
WindToFrame
CPane 357
WindToFrameR
CPane 358
WNEIsImplemented 488
WriteAll
CDataFile 266
CPictFile 382
WriteBitMap
CPNTGFile 388
WriteSome
CDataFile 266

Z

Zoom
CWindow 482